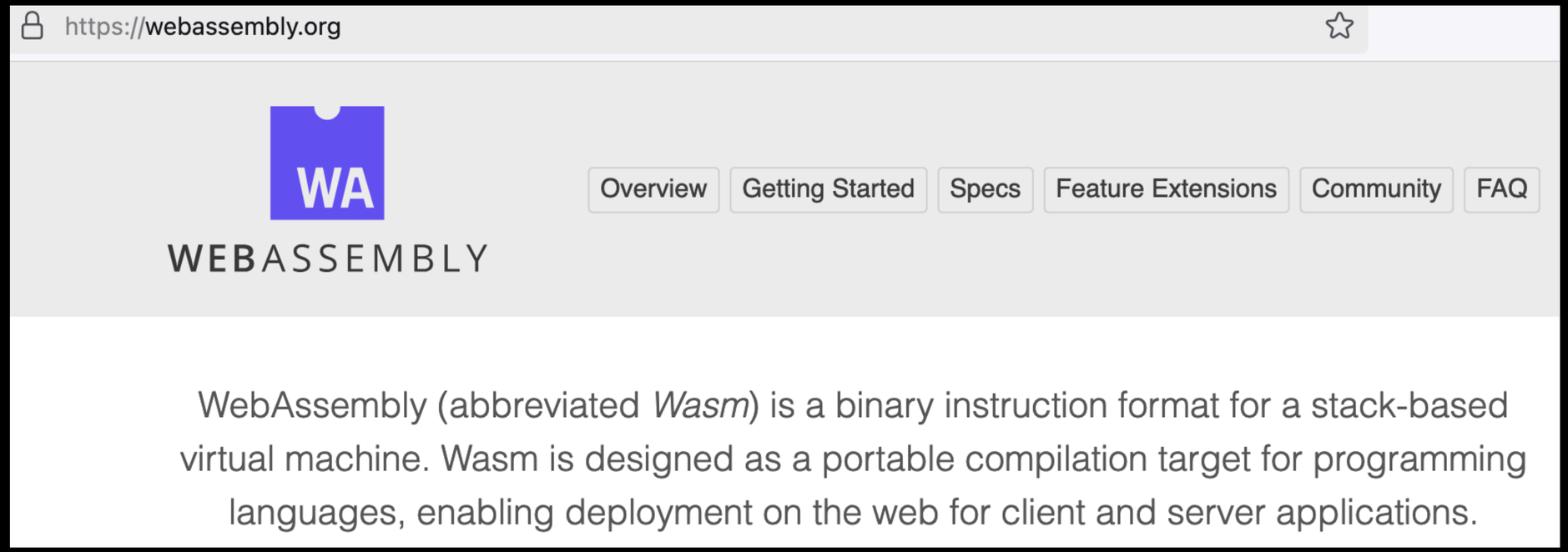
Full-Stack Correctness in Wasm

Eliminating Bugs Inside and Outside the Sandbox

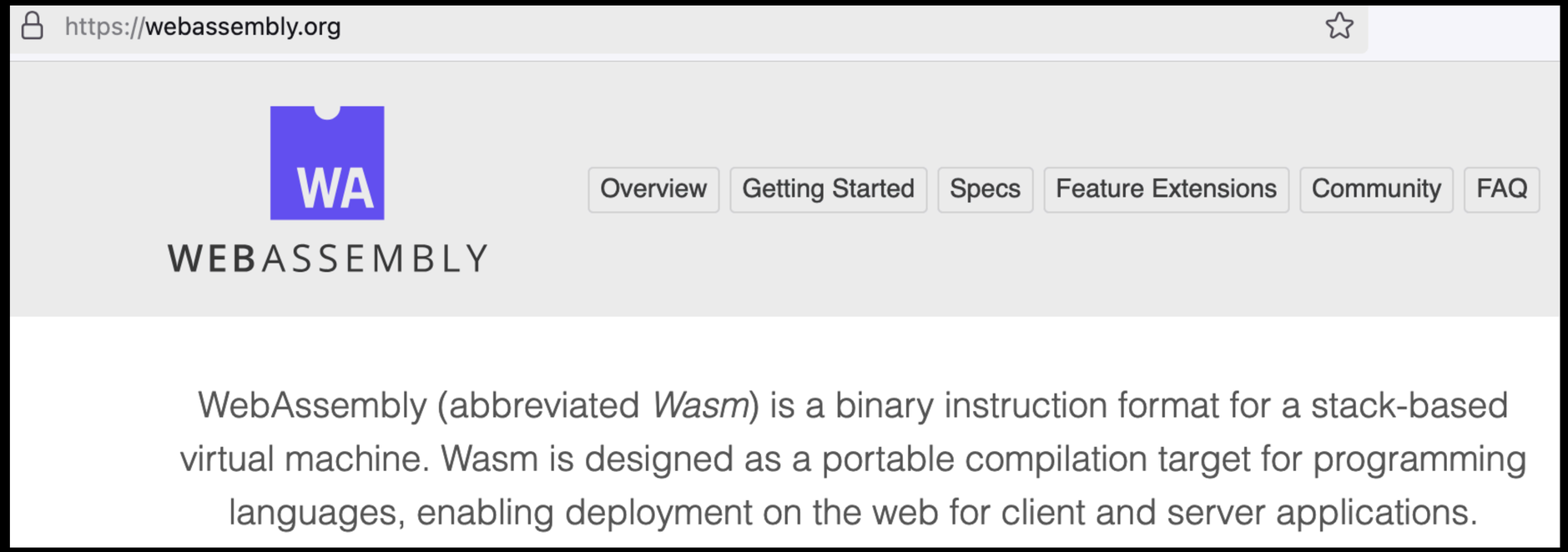
Chris Fallin ***(F5)***

Invited Talk, WAW 2025

WebAssembly is a Secure Sandbox



WebAssembly is a Secure Sandbox



Safe

WebAssembly describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines. When embedded in the web, WebAssembly will enforce the same-origin and permissions security policies of the browser.

WebAssembly is a Secure Sandbox

Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly

https://bytecodealliance.org/articles/announcing-the-bytecode-alliance

WebAssembly is a Secure Sandbox

Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly

secure by default,

encapsulated by default.

This gives us memory isolation between the two modules.

https://bytecodealliance.org/articles/announcing-the-bytecode-alliance

WebAssembly is a Secure Sandbox

Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly

secure by default,

encapsulated by default.

This gives us memory isolation between the two modules.

12x **secure**

https://bytecodealliance.org/articles/announcing-the-bytecode-alliance

WebAssembly is a Secure Sandbox

https://wasmtime/dev/

https://cranelift.dev/

WebAssembly is a Secure Sandbox

Wasmtime

A fast and secure runtime for WebAssembly

strongly focused on correctness and security.

https://wasmtime/dev/

https://cranelift.dev/

WebAssembly is a Secure Sandbox

How WebAssembly Offers Secure Development through Sandboxing

Industry experts discuss why and how WebAssembly offers developers a significantly higher security bar than previous technologies.

https://thenewstack.io/how-webassembly-offers-secure-development-through-sandboxing/

WebAssembly is a Secure Sandbox

How WebAssembly Offers Secure Development through Sandboxing

Industry experts discuss why and how WebAssembly offers developers a significantly higher security bar than previous technologies.

"It is very secure in that it uses sandboxed memory,

https://thenewstack.io/how-webassembly-offers-secure-development-through-sandboxing/

WebAssembly is a Secure Sandbox

How WebAssembly Offers Secure Development through Sandboxing

Industry experts discuss why and how WebAssembly offers developers a significantly higher security bar than previous technologies.

"It is very secure in that it uses sandboxed memory,

"Those things make WebAssembly incredibly more secure as a starting point for development."

https://thenewstack.io/how-webassembly-offers-secure-development-through-sandboxing/

WebAssembly is a Secure Sandbox

"It is very secure in that it uses sandboxed memory,

WebAssembly is a Secure Sandbox

It is very secure

WebAssembly is a Secure Sandbox

It is very secure

WebAssembly is a Secure Sandbox?

It is very secure

Wasm engine and

compiler engineers:

CVE-2021-32629

CVE-2021-32629

- April 21, 2021 was a beautiful morning in California…

CVE-2021-32629

- April 21, 2021 was a beautiful morning in California…
- “The daemon keeps segfaulting”

CVE-2021-32629

- April 21, 2021 was a beautiful morning in California…
- “The daemon keeps segfaulting” [this never happens]

CVE-2021-32629

- April 21, 2021 was a beautiful morning in California…
- “The daemon keeps segfaulting” [this never happens]
- “faults are coming from inside compiled Wasm code”

CVE-2021-32629

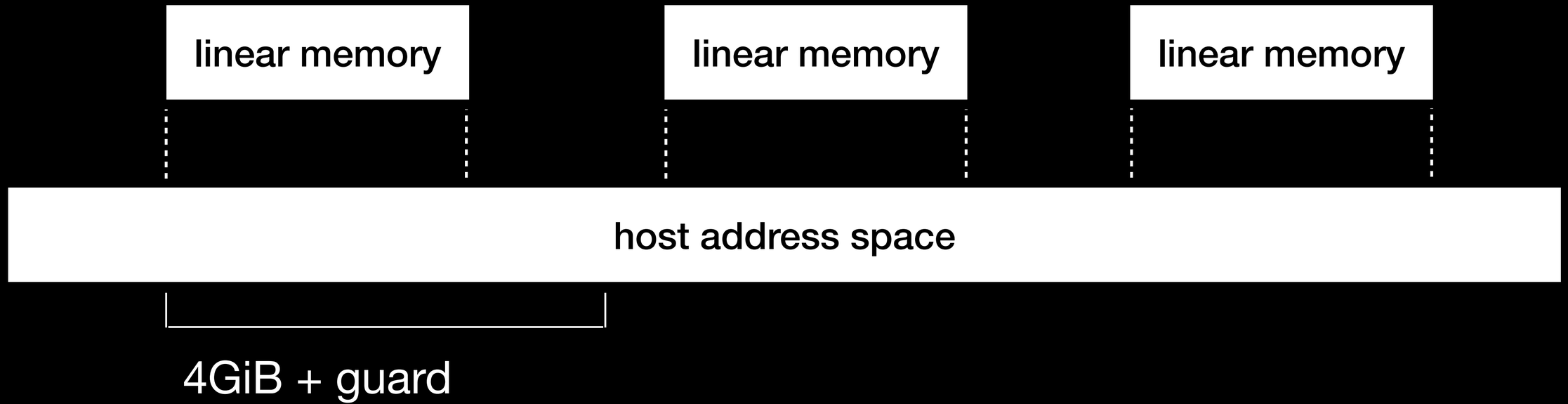
- April 21, 2021 was a beautiful morning in California…
- “The daemon keeps segfaulting” [this never happens]
- “faults are coming from inside compiled Wasm code”
- “I’m calling an incident”

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

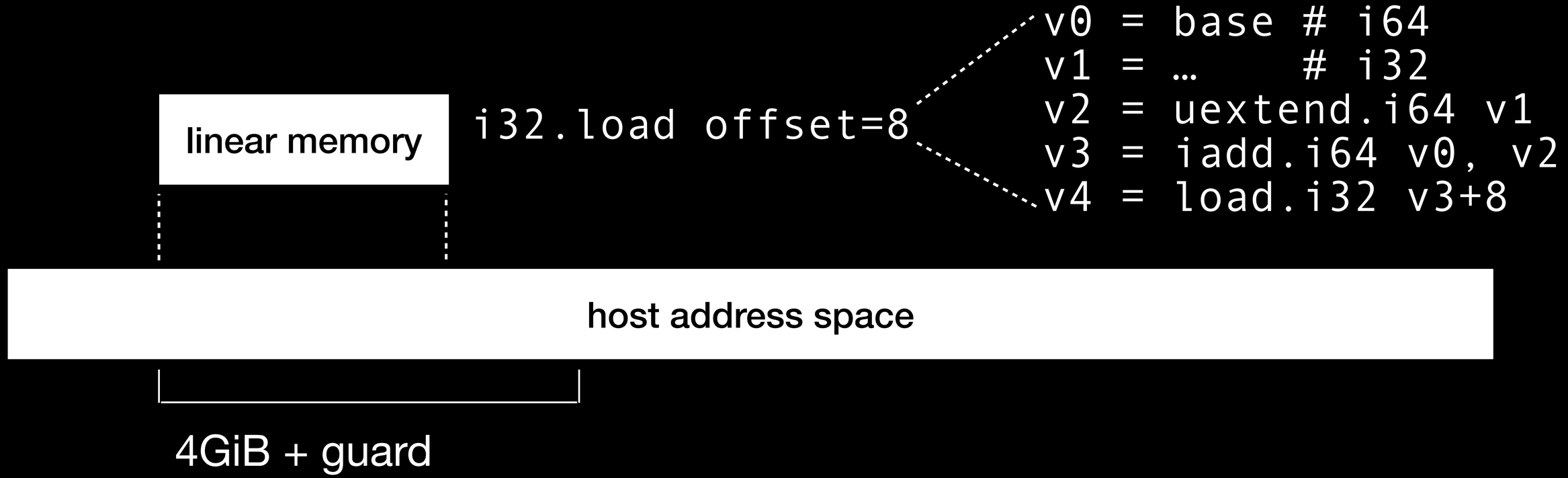


CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

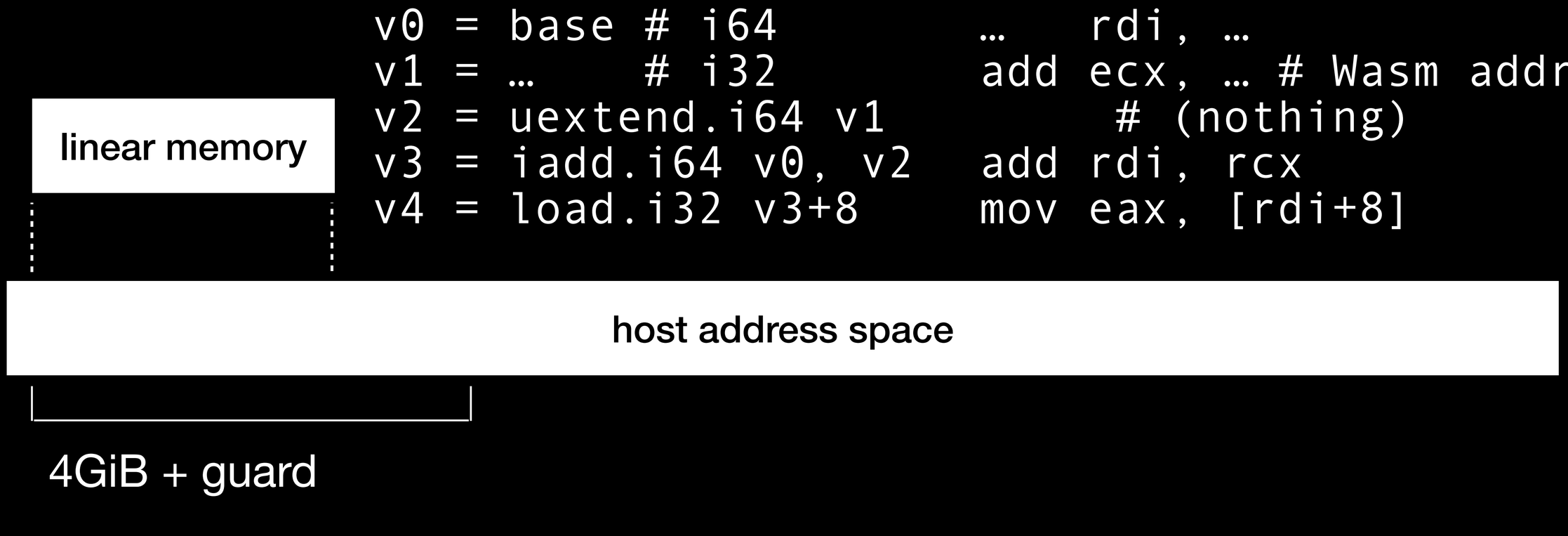
CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



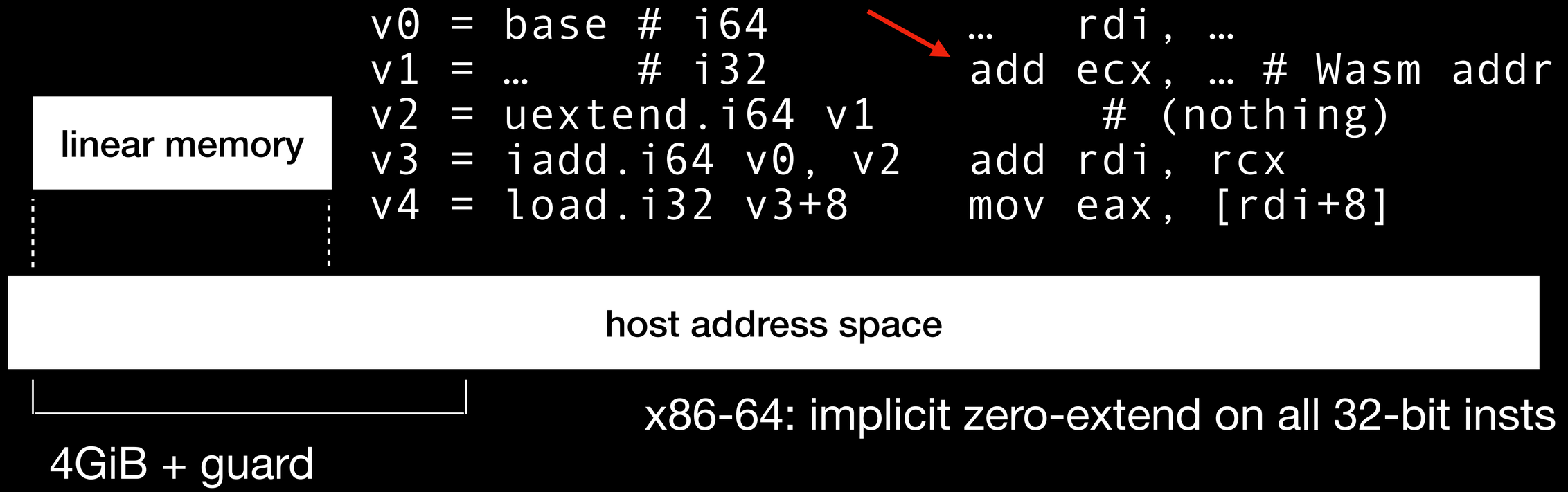
CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



CVE-2021-32629

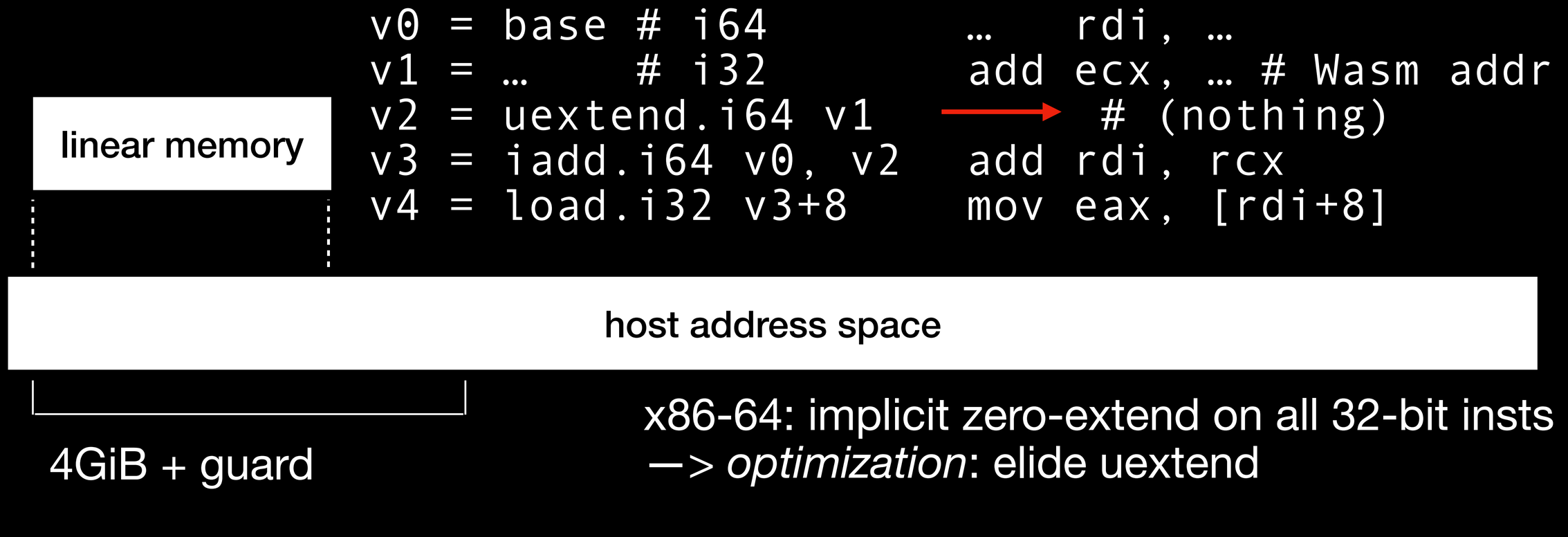
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



x86-64: implicit zero-extend on all 32-bit insts

CVE-2021-32629

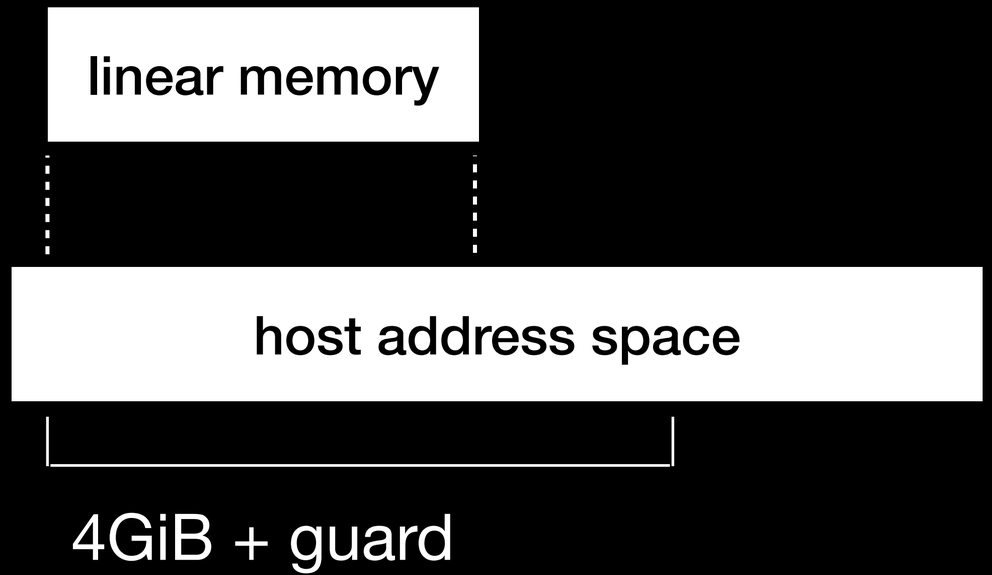
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



x86-64: implicit zero-extend on all 32-bit insts
—> *optimization*: elide uextend

CVE-2021-32629

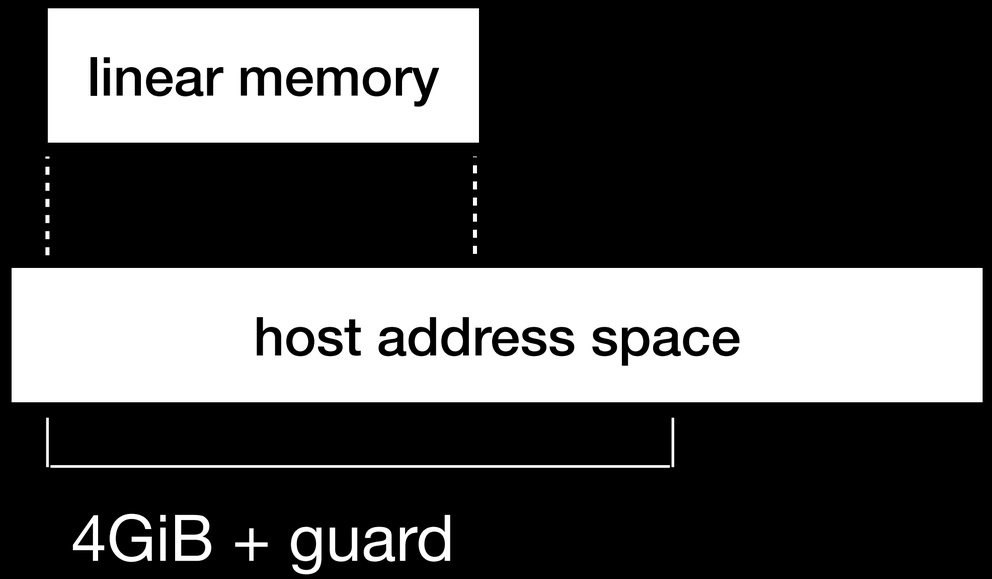
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…    rdi, …
add ecx, … # Wasm addr
      # (nothing)
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

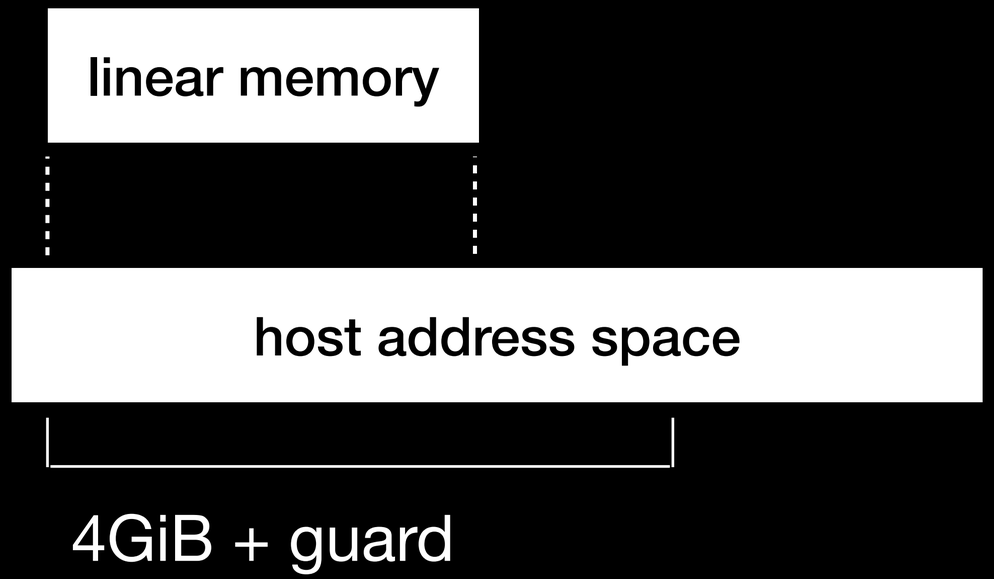
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…    rdi, …
add  ecx, … # Wasm addr
      # REGALLOC SPILL
      # …
      # REGALLOC RELOAD
add  rdi, rcx
mov  eax, [rdi+8]
```

CVE-2021-32629

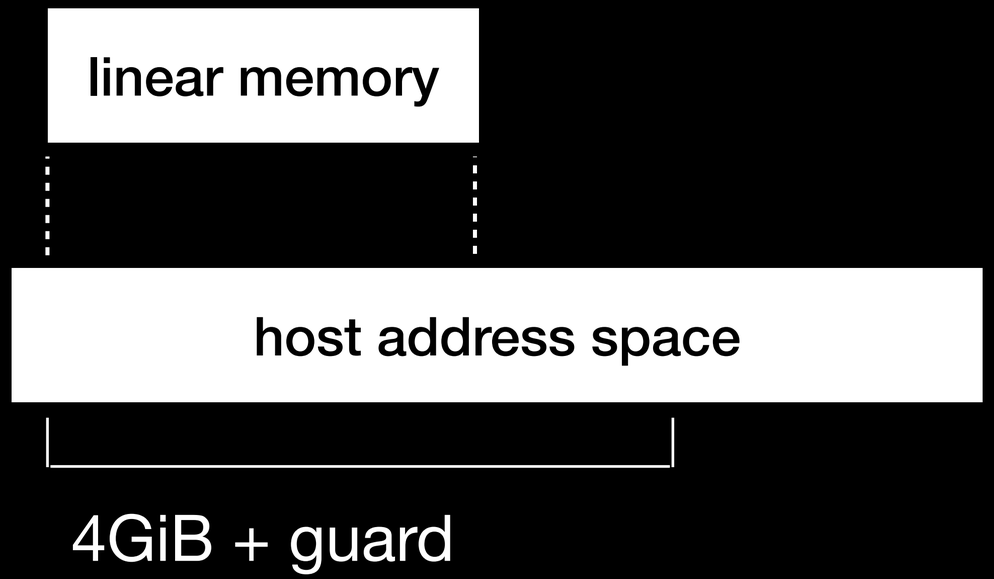
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], rcx # SPILL
     # …
mov rcx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

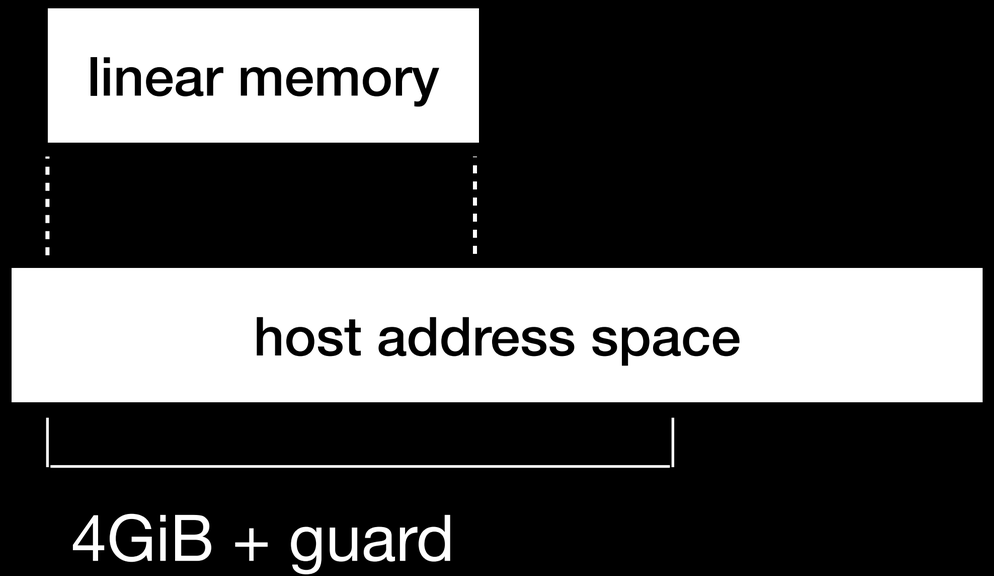



```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], rcx # SPILL
     # …
mov rcx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

Optimization: spill/reload actual value width (important for f32/f64 in 128-bit XMM regs)

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…    rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx # SPILL
     #
mov ecx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

Optimization: spill/reload actual value width (important for f32/f64 in 128-bit XMM regs)

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

1. Optimization: elide 32-to-64 zero-extends on x86-64 — use implicit dest widening

```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], rcx   # SPILL
     # …
mov rcx, [rsp+K]   # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

1. Optimization: elide 32-to-64 zero-extends on x86-64 — use implicit dest widening
2. Optimization: spill only actual value width

```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx   # SPILL
     # …
mov ecx, [rsp+K]   # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

1. Optimization: elide 32-to-64 zero-extends on x86-64 — use implicit dest widening
2. Optimization: spill only actual value width
3. Bug: use upper bits of register when technically undefined per IR->machine mapping

```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx   # SPILL
     # …
mov ecx, [rsp+K]   # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

1. Optimization: elide 32-to-64 zero-extends on x86-64 — use implicit dest widening
2. Optimization: spill only actual value width
3. Bug: use upper bits of register when technically undefined per IR->machine mapping
    —> we elided uextend but value is still narrow

```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx   # SPILL
     # …
mov ecx, [rsp+K]   # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

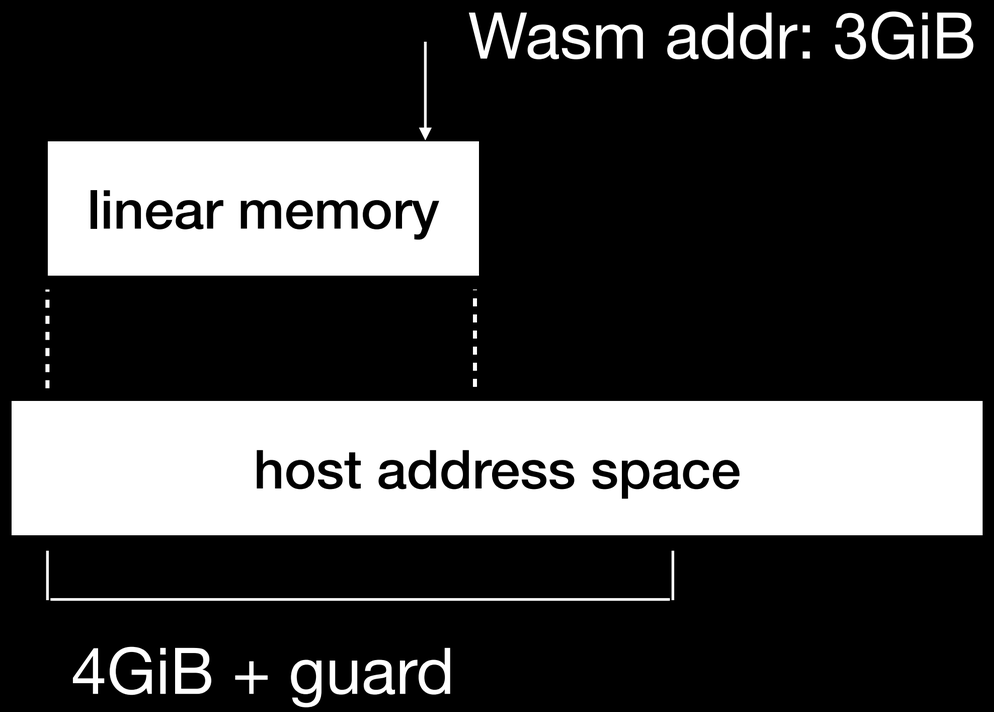
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

1. Optimization: elide 32-to-64 zero-extends on x86-64 — use implicit dest widening
2. Optimization: spill only actual value width
3. Bug: use upper bits of register when technically undefined per IR->machine mapping
4. Questionable choice: sign-extend on reload??

```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx   # SPILL
     # …
movsx rcx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

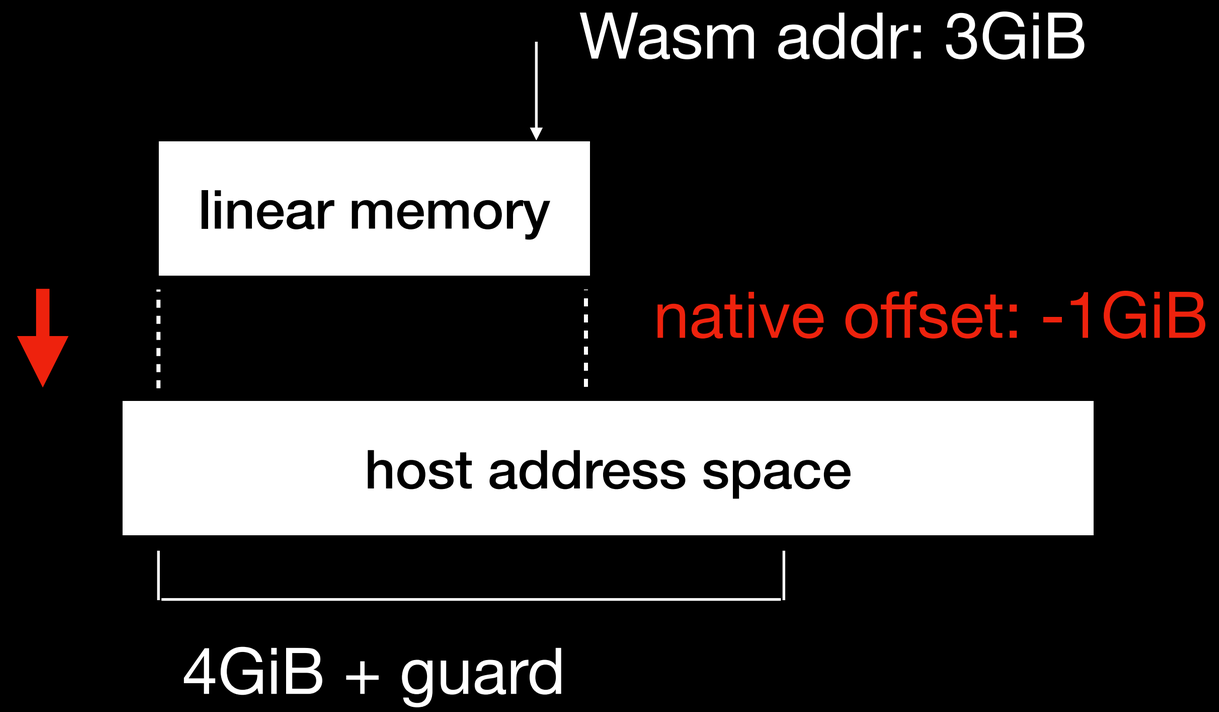
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…   rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx   # SPILL
     # …
movsx rcx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

CVE-2021-32629

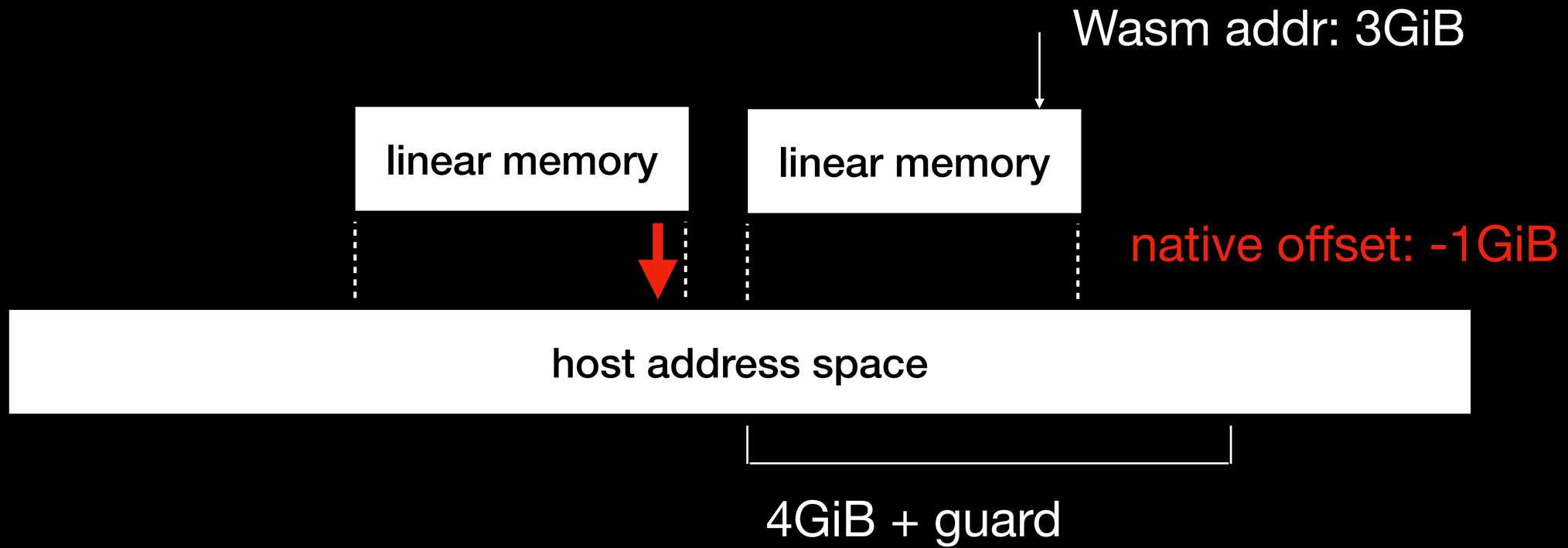
- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)




```
…    rdi, …
add ecx, … # Wasm addr
mov [rsp+K], ecx    # SPILL
      # …
movsx rcx, [rsp+K] # RELOAD
add rdi, rcx
mov eax, [rdi+8]
```

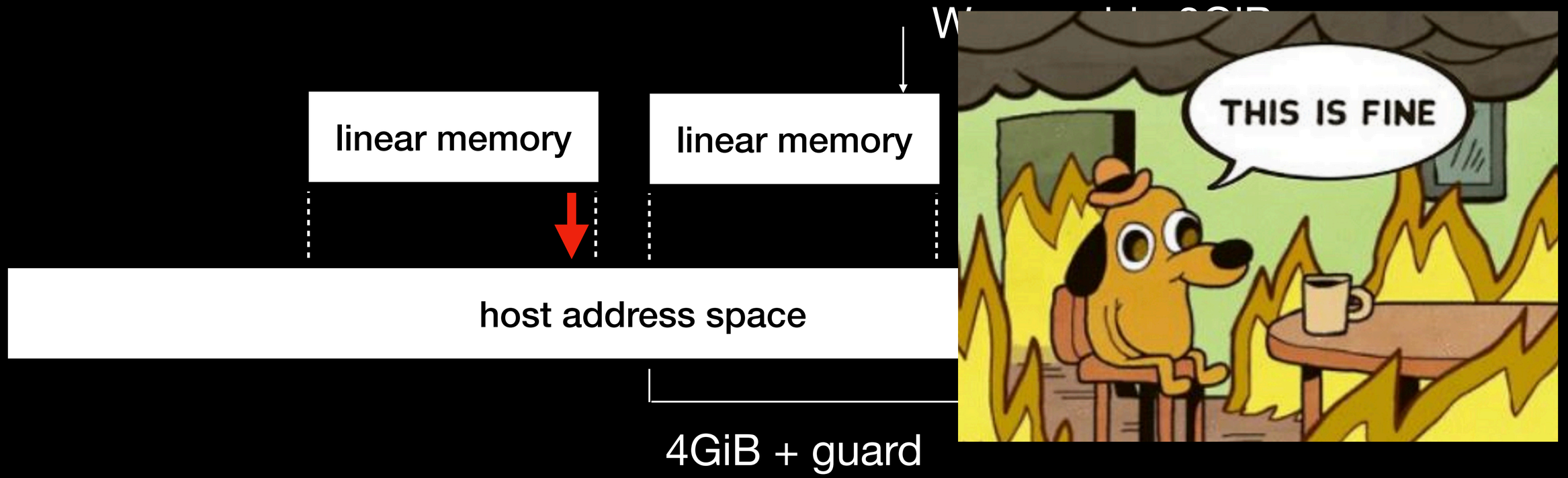
CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)



CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)
- Aftermath: emergency version bump internally; patch release; forcing function to develop our CVE release process in BA / Wasmtime (since exercised more!)

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)
- Aftermath: emergency version bump internally; patch release; forcing function to develop our CVE release process in BA / Wasmtime (since exercised more!)
- How can we avoid ever having this problem again?

CVE-2021-32629

- **Summary**: a miscompilation could result in a Wasm instance accessing memory addresses 2GiB prior to its linear memory in host address space (!)
- Aftermath: emergency version bump internally; patch release; forcing function to develop our CVE release process in BA / Wasmtime (since exercised more!)
- How can we avoid ever having this problem again*?

CVE-2023-26489

CVE-2023-26489

- Aside: it did happen again, two years later
- Summary: base + uextend(index << 3) folded to base + uextend(index) << 3 in x86-64 addressing mode selection; reach up to 34GiB beyond a memory

CVE-2023-26489

- Aside: it did happen again, two years later
- Summary: base + uextend(index << 3) folded to base + uextend(index) << 3 in x86-64 addressing mode selection; reach up to 34GiB beyond a memory
- One must imagine ~~Sisyphus~~ verification researchers happy

SFI and Trusting Compilers

Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

One way to provide fault isolation among cooperating software modules is to place each in its own address space. However, for tightly-coupled modules, this solution incurs prohibitive context switch overhead. In this paper, we present a software approach to implementing fault isolation within a single address space.

SOSP 1993

SFI and Trusting Compilers

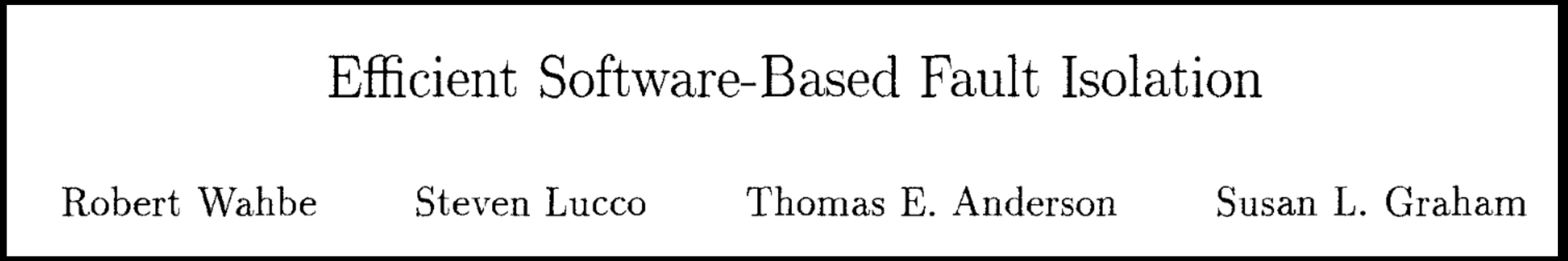
Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

SOSP 1993

- We put many instances in a single address space, and add *software* checks inline, to enable fast context switching — essential for many workloads!

SFI and Trusting Compilers



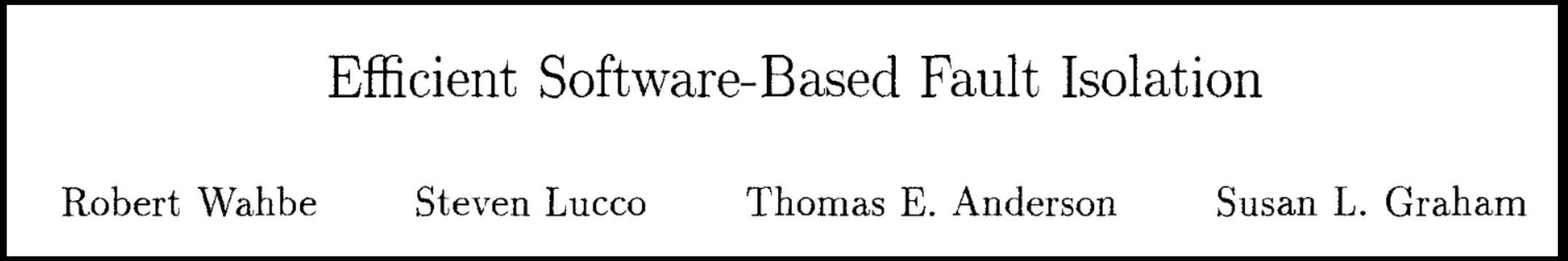
Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

SOSP 1993

- We put many instances in a single address space, and add *software* checks inline, to enable fast context switching — essential for many workloads!
  - Browser: fast Wasm-to-JS interaction (~native func call) on one webpage
  - Server-side: extremely dense multi-tenant environments (timeslicing)

SFI and Trusting Compilers

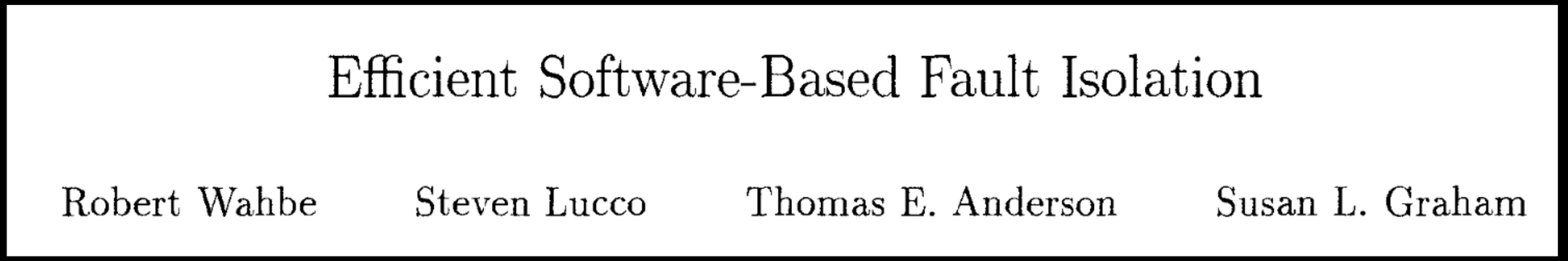

Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

SOSP 1993

- We put many instances in a single address space, and add *software* checks inline, to enable fast context switching — essential for many workloads!
  - Browser: fast Wasm-to-JS interaction (~native func call) on one webpage
  - Server-side: extremely dense multi-tenant environments (timeslicing)
- This is Wasm's secret superpower (tiny sandboxes — nanoprocesses)

SFI and Trusting Compilers



SOSP 1993

- We put many instances in a single address space, and add *software* checks inline, to enable fast context switching — essential for many workloads!
  - Browser: fast Wasm-to-JS interaction (~native func call) on one webpage
  - Server-side: extremely dense multi-tenant environments (timeslicing)
- This is Wasm's secret superpower (tiny sandboxes — nanoprocesses)
- But we ***must trust the compiler***

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8
  - Fuzzing with symbolic translation validation of register allocation

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8
  - Fuzzing with symbolic translation validation of register allocation
  - “Chaos testing” in compiler pipeline

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8
  - Fuzzing with symbolic translation validation of register allocation
  - “Chaos testing” in compiler pipeline
  - trial by fire in real world: Cranelift as rustc backend

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8
  - Fuzzing with symbolic translation validation of register allocation
  - “Chaos testing” in compiler pipeline
  - trial by fire in real world: Cranelift as rustc backend
- Somehow these CVEs still happen occasionally (~0.5 per year)

How to Write a Correct Compiler

- We do *a lot* to try to ensure correctness
  - Differential fuzzing against: Wasm spec interp, wasmi, V8
  - Fuzzing with symbolic translation validation of register allocation
  - “Chaos testing” in compiler pipeline
  - trial by fire in real world: Cranelift as rustc backend
- Somehow these CVEs still happen occasionally (~0.5 per year)
- “I would simply prove the compiler correct”

How to Write a Correct Compiler

- “I would simply prove the compiler correct”

How to Write a Correct Compiler

- “I would simply prove the compiler correct”

✅ *Challenge Accepted*

Anti-Goal

THE COMPCERT C COMPILER

Download CompCert C Read the manual

CompCert C is a compiler for the C programming language. Its intended use is the compilation of life-critical and mission-critical software written in C and meeting high levels of assurance. It accepts most of the ISO C 99 language, with some exceptions and a few extensions. It produces machine code for the PowerPC, ARM, RISC-V and x86 (32 and 64 bits) architectures. Performance of the generated code is decent but not outstanding: on PowerPC, about 90% of the performance of GCC version 4 at optimization level 1.

What sets CompCert C apart from any other production compiler, is that it is *formally verified*, using machine-assisted mathematical proofs, to be exempt from *miscompilation* issues. In other words, the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This level of confidence in the

Anti-Goal

THE COMPCERT C COMPILER

Download CompCert C Read the m

CompCert C is a compiler for the C programming la
compilation of life-critical and mission-critical soft
levels of assurance. It accepts most of the ISO C 9
few extensions. It produces machine code for the
bits) architectures. Performance of the generated
PowerPC, about 90% of the performance of GCC ve

What sets CompCert C apart from any other produ
verified, using machine-assisted mathematical pro
issues. In other words, the executable code it proc
specified by the semantics of the source C progran

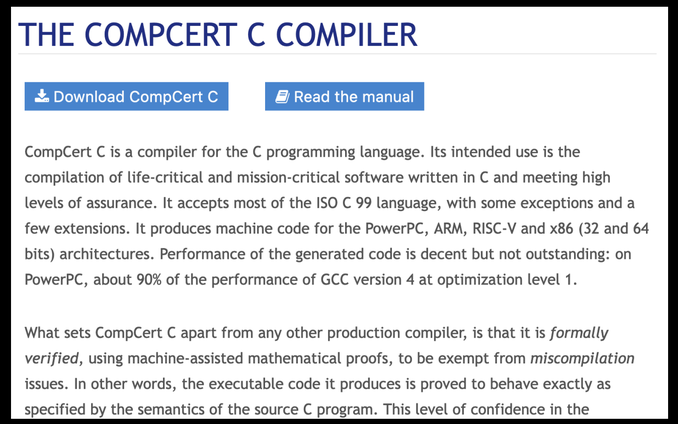
CAKEML

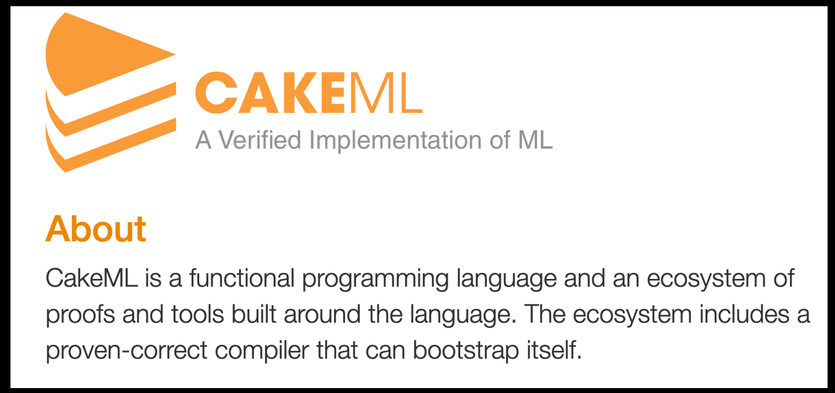
A Verified Implementation of ML

About

CakeML is a functional programming language and an ecosystem of proofs and tools built around the language. The ecosystem includes a proven-correct compiler that can bootstrap itself.

Anti-Goal





- Engineered from scratch for verification (we have ~200KLoC existing code)
- Optimizations limited by provability (we don’t want to limit perf *too* much)
- Enormous manual effort (we’re a tiny team and verification is one of many demands on us; can’t afford ~engineer-century of work)

Potential Goals?

- Can we verify a part of our compiler (where bugs are more common) more thoroughly?

- Can we verify limited properties of the code (e.g. linear memory sandboxing) end-to-end?

Potential Goals?

- Can we verify a part of our compiler (where bugs are more common) more thoroughly?

→ *SMT on instruction selector rules (ASPLOS'24)*

- Can we verify limited properties of the code (e.g. linear memory sandboxing) end-to-end?

→ *Proof-carrying code (ongoing)*

Outline

- Formal Verification in Instruction Selection
- Proof-Carrying Code for Sandboxing Logic
- Guest-Code Correctness

Instruction Lowering Verification

BA RFC 15: ISLE instruction-selection (pattern-matching) DSL, Aug 2021

Discussion: Future Implications for Verification

Though we have not yet worked out all the details, we are confident that the translation of rules expressed in the ISLE DSL into some machine-readable form for formal verification efforts should be possible. This is primarily because of the "equivalent-value" semantics that are inherent in a term-rewriting system. The denotational value of a term is the symbolic or concrete value produced by the instruction it represents (depending on the interpretation); so "all" we have to do is to write, e.g., pre/post-conditions for some SMT-solver or theorem-prover that describe the semantics of instruction terms on either side of the translation.

Instruction Lowering Verification

BA RFC 18: Cranelift roadmap for 2022 (Dec 2021)

> In the next year we should attempt to find some concrete ways to achieve formal verification of some part of the compiler. The instruction lowerings are the obvious choice, now that we have ISLE.

Instruction Lowering Verification

- Dec 2021: contact from both Alexa VanHattum and Fraser Brown
  (+ Alexa's advisor Adrian Sampson and Fraser's student Monica Pardeshi)

Instruction Lowering Verification

- Dec 2021: contact from both Alexa VanHattum and Fraser Brown
  (+ Alexa's advisor Adrian Sampson and Fraser's student Monica Pardeshi)

✅ *Academic collaboration acquired; let's go!*

Instruction Lowering Verification

Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection

Alexa VanHattum
Wellesley College
Wellesley, MA, USA
av111@wellesley.edu

Monica Pardeshi
Carnegie Mellon University
Pittsburgh, PA, USA
mpardesh@andrew.cmu.edu

Chris Fallin
Fastly
San Francisco, CA, USA
cfallin@fastly.com

Adrian Sampson
Cornell University
Ithaca, NY, USA
asampson@cs.cornell.edu

Fraser Brown
Carnegie Mellon University
Pittsburgh, PA, USA
fraserb@andrew.cmu.edu

ASPLOS 2024

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```

Cranelift IR (CLIF)

rotr (rotate right)

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```

Cranelift IR (CLIF)

rotr (rotate right)

aarch64 machine code

rotr (rotate right)

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```

Cranelift IR (CLIF)

rotr (rotate right)

SMT (theory of bitvectors)

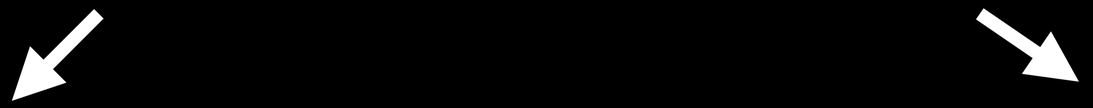
aarch64 machine code

rotr (rotate right)

SMT (theory of bitvectors)

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```



Cranelift IR (CLIF)

rotr (rotate right)

SMT (theory of bitvectors)

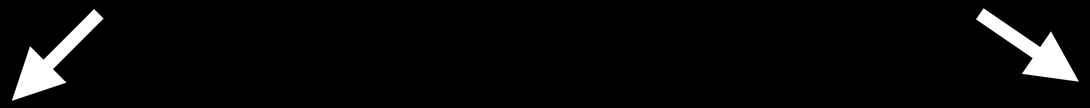
aarch64 machine code

rotr (rotate right)

SMT (theory of bitvectors)

Instruction Lowering Verification

```
1 (rule
2   (lower (rotr x y))
3   (a64_rotr I64 x y))
```



Cranelift IR (CLIF)

rotr (rotate right)

SMT (theory of bitvectors)

(or counterexample)

aarch64 machine code

rotr (rotate right)

SMT (theory of bitvectors)

Instruction Lowering Verification

```
(lower (has_type $I64 (rotr x y)) …)
```

Instruction Lowering Verification

```
lower
   has_type
       value_def
           InstructionData.BinaryOp (Op.Rotr)
```

Instruction Lowering Verification

```
lower
   has_type
      value_def
         InstructionData.BinaryOp (Op.Rotr)

(a64_rotr x y)
```

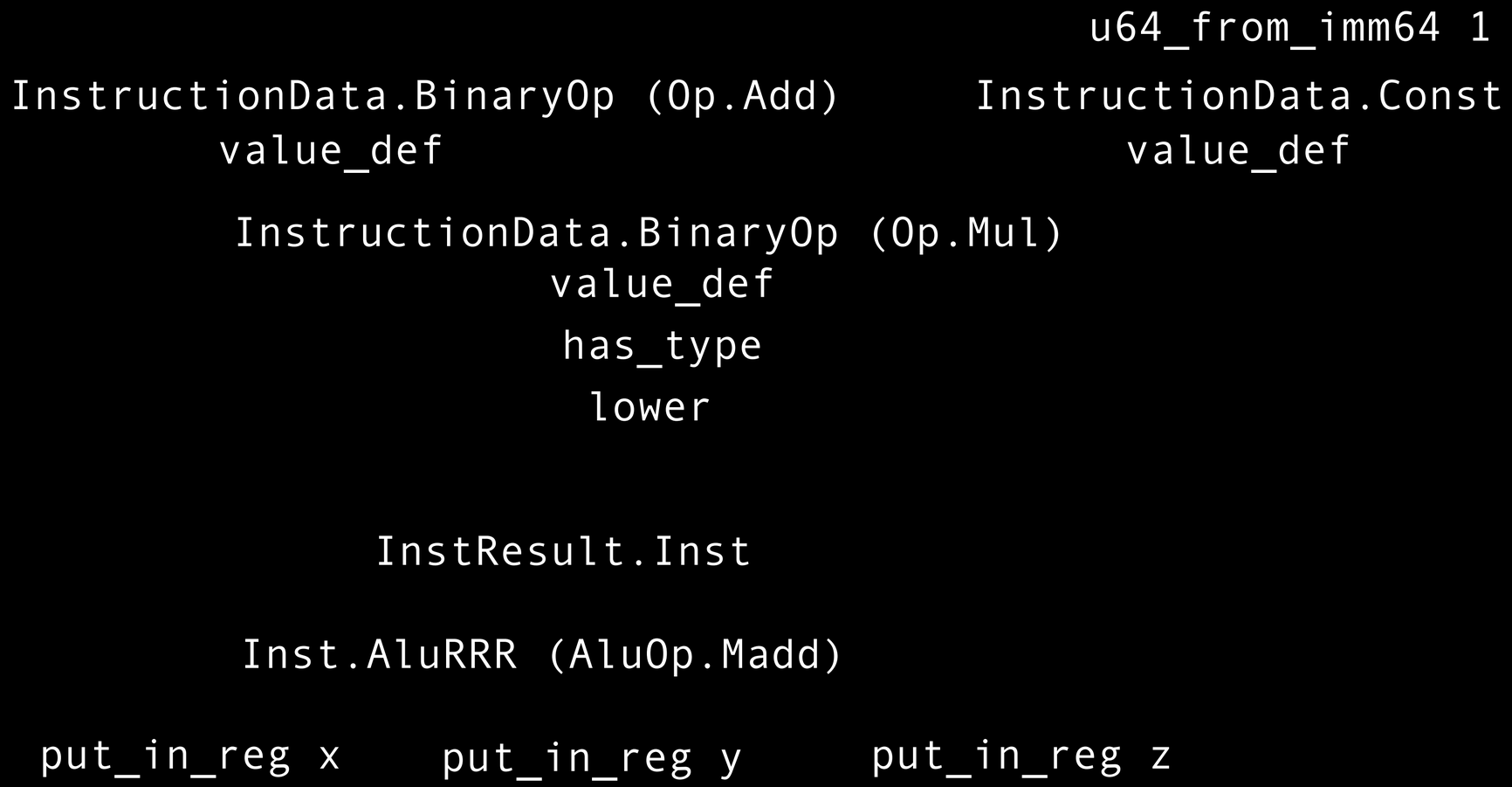
Instruction Lowering Verification

```
lower
  has_type
    value_def
      InstructionData.BinaryOp (Op.Rotr)

InstResult.Inst
  Inst.AluRRR (AluOp.Rotr)
```

Instruction Lowering Verification

```
lower
  has_type
    value_def
      InstructionData.BinaryOp (Op.Rotr)

InstResult.Inst
  Inst.AluRRR (AluOp.Rotr)
    put_in_reg x
    put_in_reg y
```

Instruction Lowering Verification



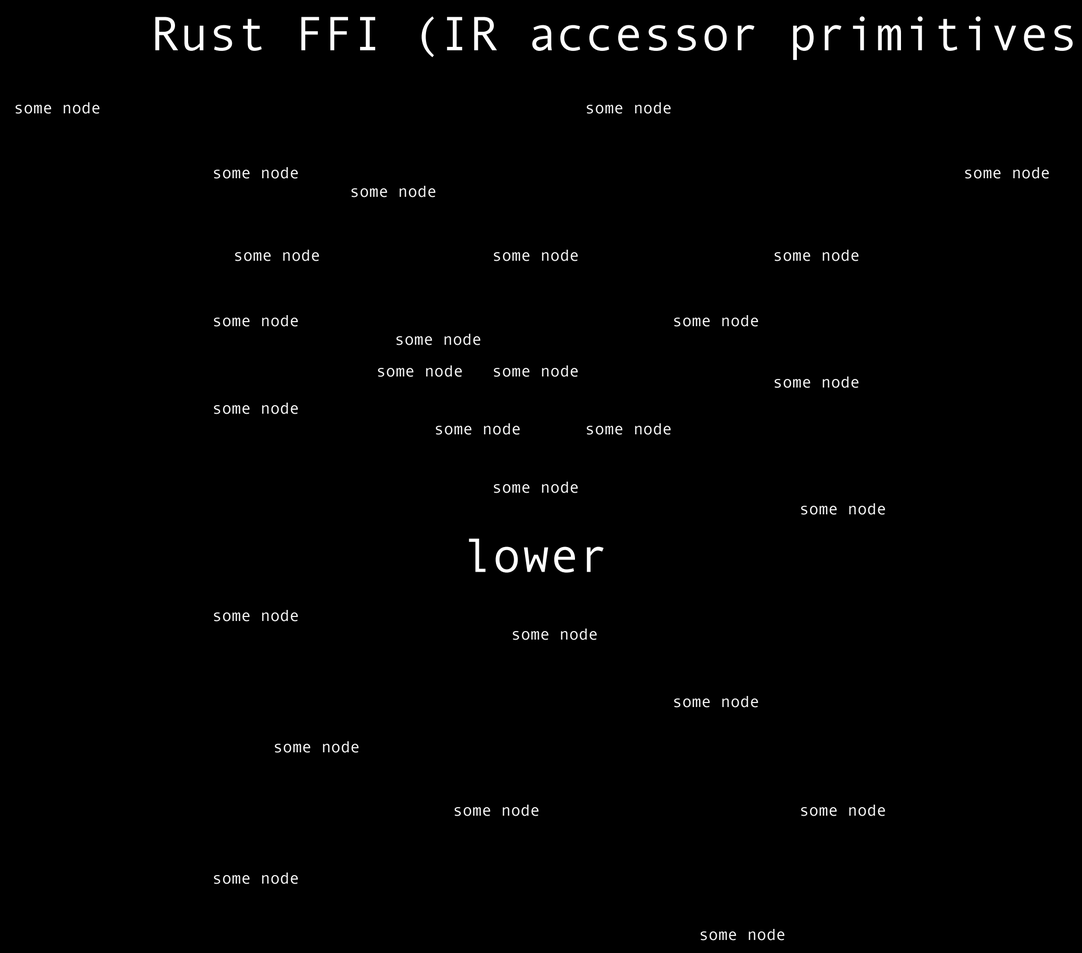
Instruction Lowering Verification

some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node
some node

lower

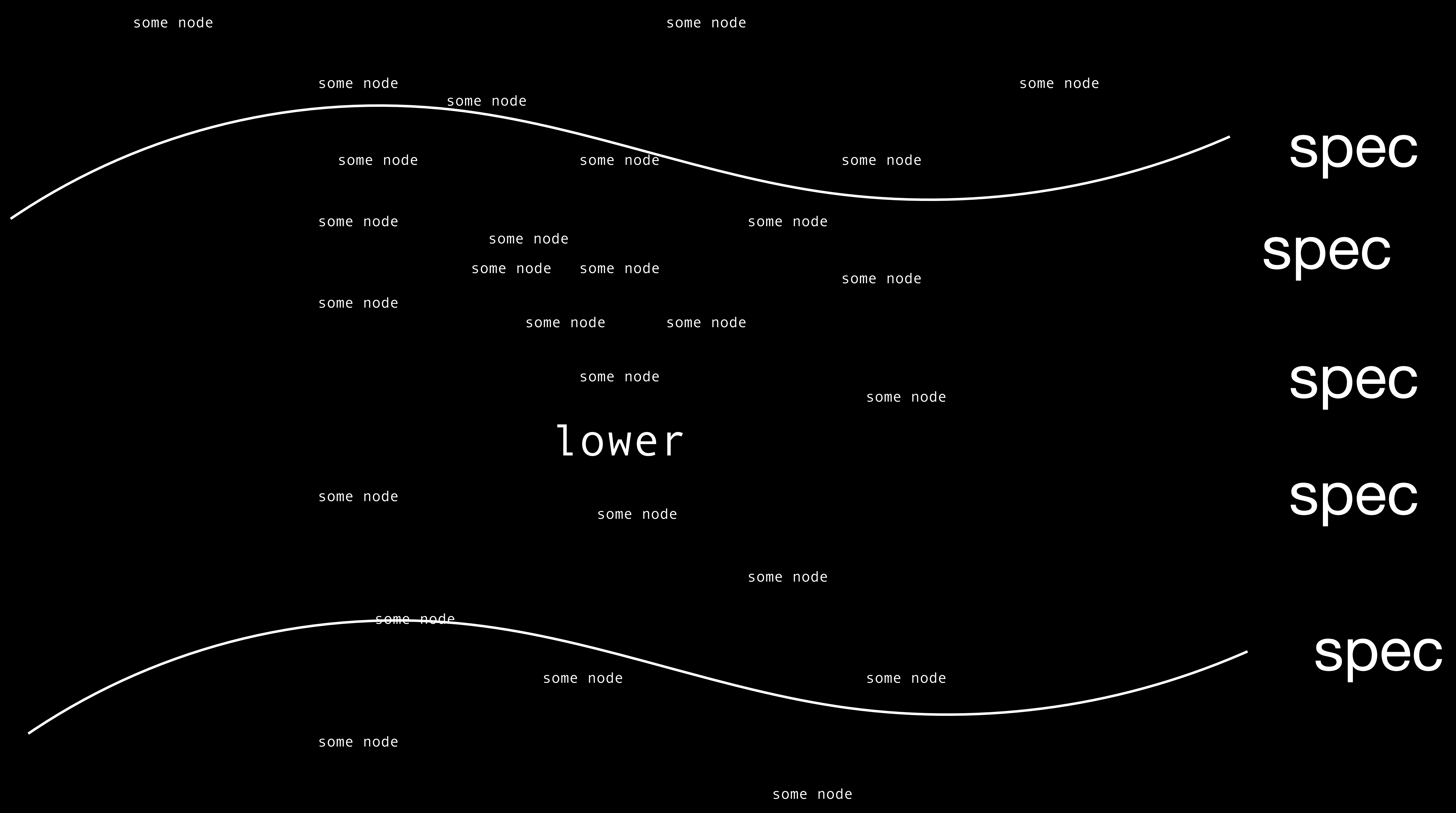
some node
some node
some node
some node
some node
some node
some node
some node
some node

Instruction Lowering Verification



Instruction Lowering Verification

Rust FFI (IR accessor primitives)



Rust FFI (instruction emit primitives)

Instruction Lowering Verification

```
2321      (spec (cmp ty x y)
2322        (provide (= result (subs ty x y)))
2323        (require
2324           (or (= ty 32) (= ty 64))))
```

cranelift/codegen/src/isa/aarch64/inst.isle

Instruction Lowering Verification

- Lots more to actually make this work!
  - Type-polymorphism in rules —> “instantiate” at concrete widths
  - Type-inference to use narrower bitvectors
  - Full system of specifying “model domain” values for ISLE values
  - Good ergonomics around showing counterexamples

Instruction Lowering Verification

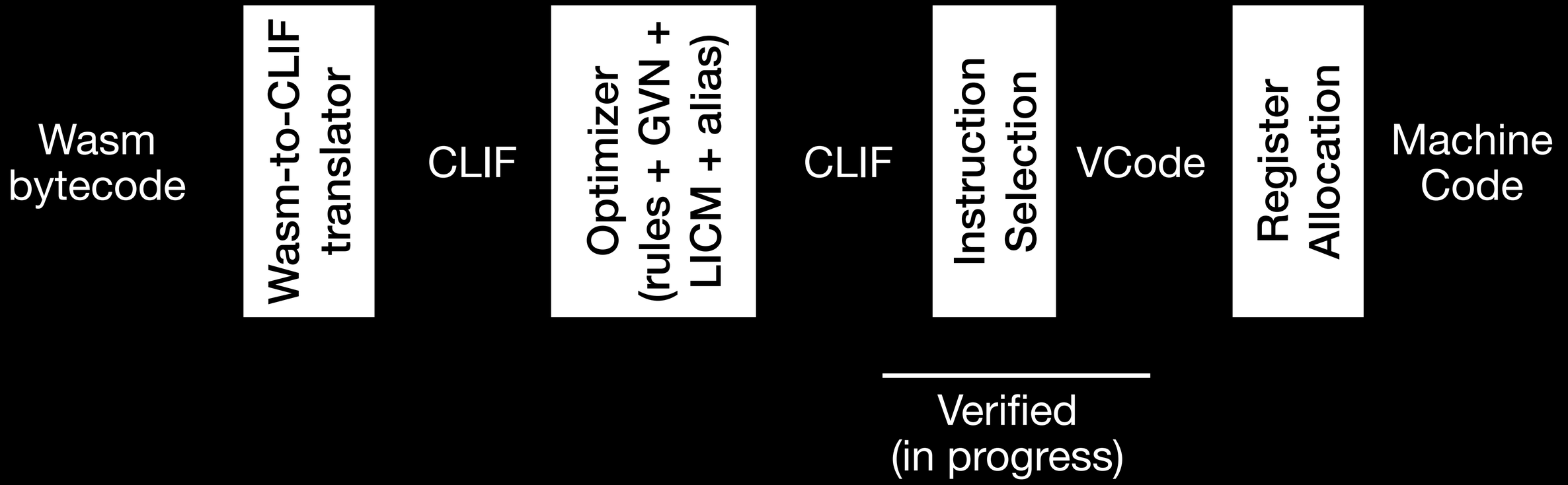
- It finds real bugs
  - Reproduced x86-64 amode bug (CVE-2023-26489)
  - Arithmetic edge cases in divides, count-leading-sign of narrow values, boolean simplification rules, …
  - Real counterexamples are invaluable
  - Ongoing extension work (especially: tying to real ISA semantics)
  - Ongoing discussions on how to integrate into our workflow to *keep* verified

Instruction Lowering Verification

- But… can we verify *something* end-to-end?

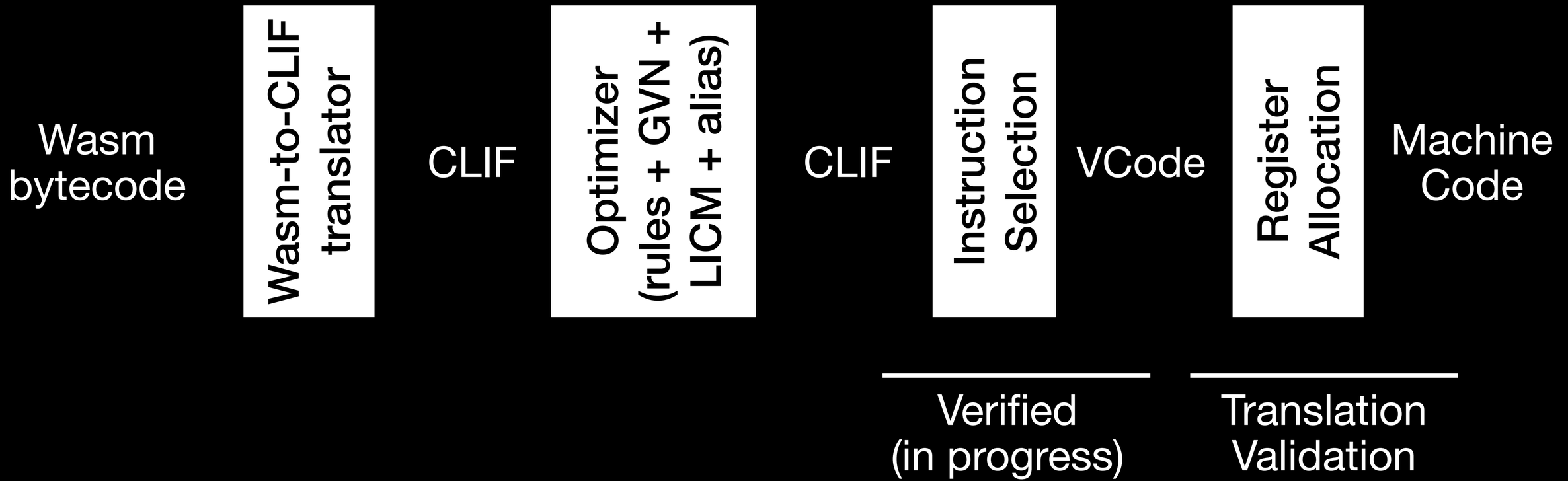
Instruction Lowering Verification

- But… can we verify *something* end-to-end?



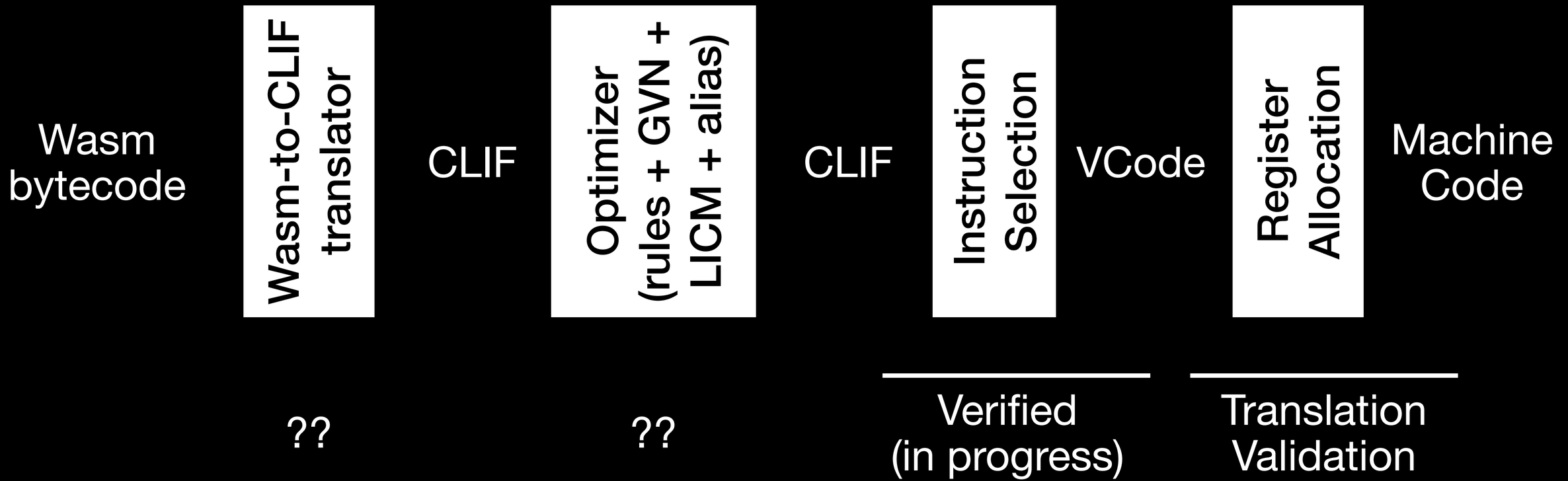
Instruction Lowering Verification

- But… can we verify *something* end-to-end?



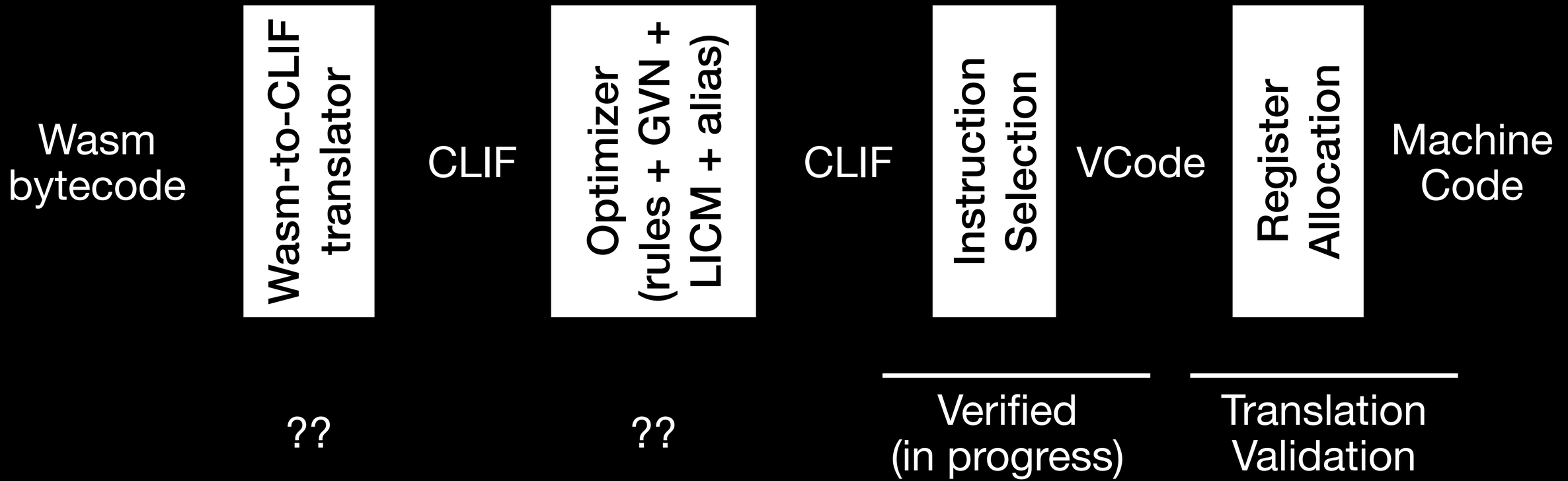
Instruction Lowering Verification

- But… can we verify *something* end-to-end?



Instruction Lowering Verification

- But… can we verify *something* end-to-end?



Sandbox Verification, End-to-End

- “Prove the compiler correct” is *Hard*(tm)

Sandbox Verification, End-to-End

- “Prove the compiler correct” is *Hard*(tm)
- Can we prove that all memory accesses in machine code access Wasm memories with valid bounds-checking (or other internal VM data)?

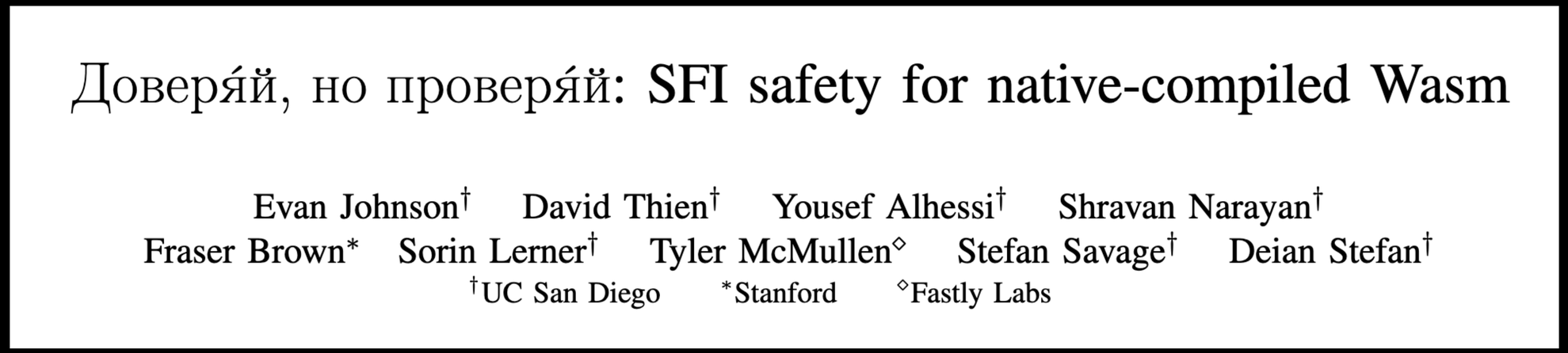
Sandbox Verification, End-to-End

- “Prove the compiler correct” is *Hard*(tm)
- Can we prove that all memory accesses in machine code access Wasm memories with valid bounds-checking (or other internal VM data)?
- Build an *independent checker* that operates on machine code: compiler no longer in the TCB (!)

Sandbox Verification, End-to-End

- “Prove the compiler correct” is *Hard*(tm)
- Can we prove that all memory accesses in machine code access Wasm memories with valid bounds-checking (or other internal VM data)?
- Build an *independent checker* that operates on machine code: compiler no longer in the TCB (!)

NDSS 2021



Доверя́й, но проверя́й: SFI safety for native-compiled Wasm

Evan Johnson[†] David Thien[†] Yousef Alhessi[†] Shravan Narayan[†]
Fraser Brown[*] Sorin Lerner[†] Tyler McMullen[◇] Stefan Savage[†] Deian Stefan[†]
[†]UC San Diego [*]Stanford [◇]Fastly Labs

Sandbox Verification, End-to-End

NDSS 2021

Доверя́й, но проверя́й: SFI safety for native-compiled Wasm

Evan Johnson† David Thien† Yousef Alhessi† Shravan Narayan†
Fraser Brown* Sorin Lerner† Tyler McMullen◇ Stefan Savage† Deian Stefan†
†UC San Diego *Stanford ◇Fastly Labs

- It even operates on Cranelift!

Sandbox Verification, End-to-End

NDSS 2021

Доверя́й, но проверя́й: SFI safety for native-compiled Wasm

Evan Johnson† David Thien† Yousef Alhessi† Shravan Narayan†
Fraser Brown* Sorin Lerner† Tyler McMullen◇ Stefan Savage† Deian Stefan†
†UC San Diego *Stanford ◇Fastly Labs

- It even operates on Cranelift!
- … but not Wasmtime (older Lucet runtime)

Sandbox Verification, End-to-End

NDSS 2021

Доверя́й, но проверя́й: SFI safety for native-compiled Wasm

Evan Johnson† David Thien† Yousef Alhessi† Shravan Narayan†
Fraser Brown* Sorin Lerner† Tyler McMullen◇ Stefan Savage† Deian Stefan†
†UC San Diego *Stanford ◇Fastly Labs

- It even operates on Cranelift!
- … but not Wasmtime (older Lucet runtime)
- … and on the output of a much older (poorly optimizing) Cranelift

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
  - Key idea: lattice-based abstract interpretation over machine registers

```
# rdi: heap base
add eax, …    # an i32 Wasm address
mov rbx, [rdi+rax+0x100]
```

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
  - Key idea: lattice-based abstract interpretation over machine registers

```
# rdi: heap base
add eax, …    # an i32 Wasm address
mov rbx, [rdi+rax+0x100]
```

rdi: HeapBase

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
  - Key idea: lattice-based abstract interpretation over machine registers

```
# rdi: heap base
add eax, …    # an i32 Wasm address
mov rbx, [rdi+rax+0x100]
```

rdi: HeapBase

rax: Bounded4GB

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
  - Key idea: lattice-based abstract interpretation over machine registers

```
# rdi: heap base
add eax, …    # an i32 Wasm address
mov rbx, [rdi+rax+0x100]
```

rdi: HeapBase

rax: Bounded4GB

access to
HeapBase + Bounded4GB ->
valid heap address

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Prototyped after 2021 CVE for *limited* domain (one memory, no dynamic bounds checking)
  - 30% compile-time overhead

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Prototyped after 2021 CVE for *limited* domain (one memory, no dynamic bounds checking)
  - 30% compile-time overhead
  - What about 2023, and full production feature support?

VeriWasm

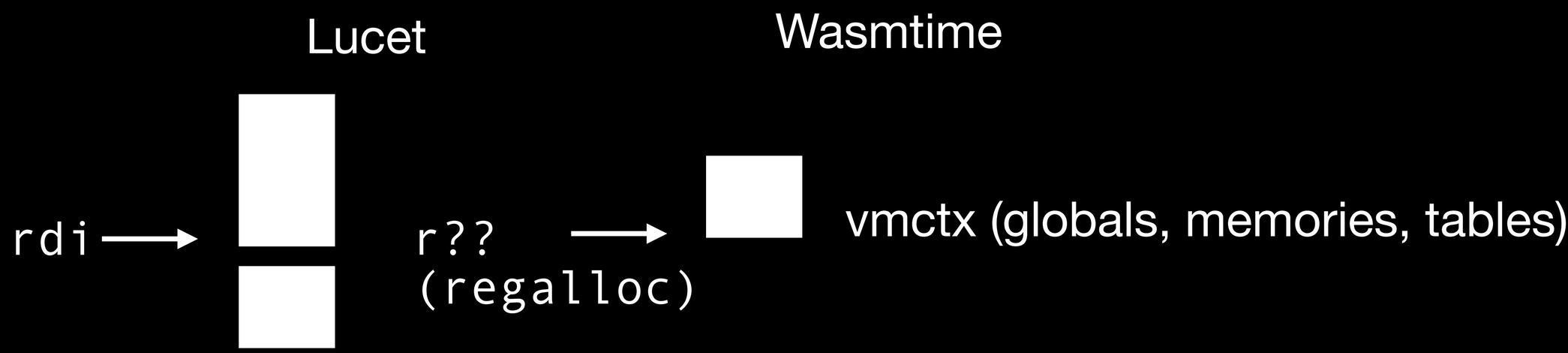
- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables

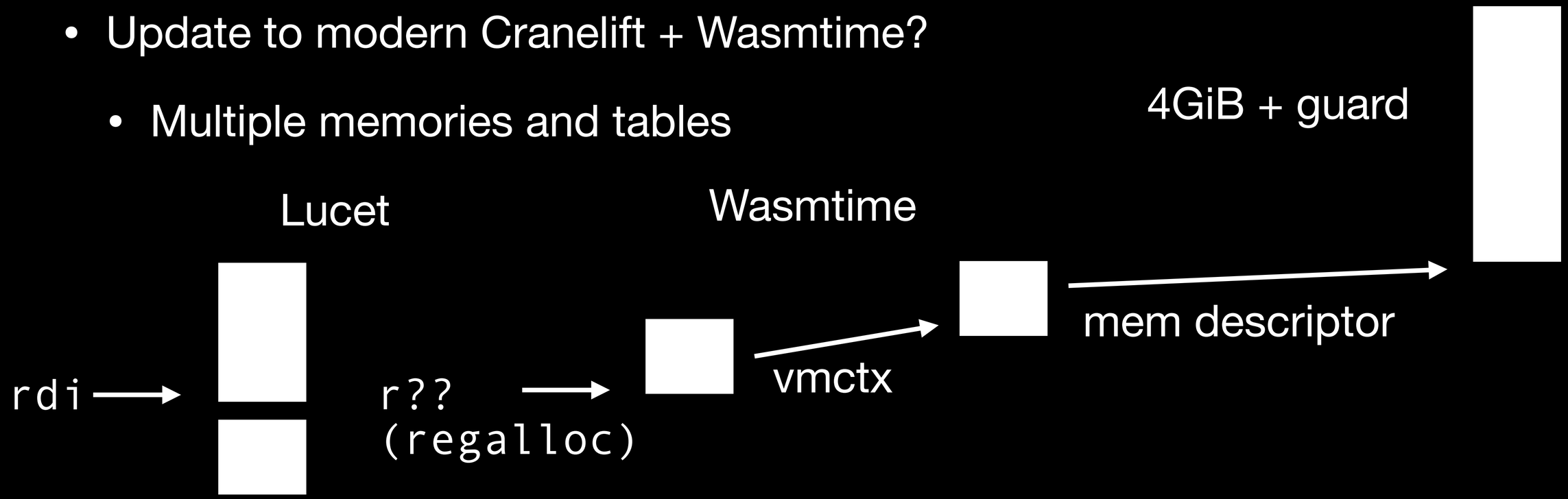
VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables



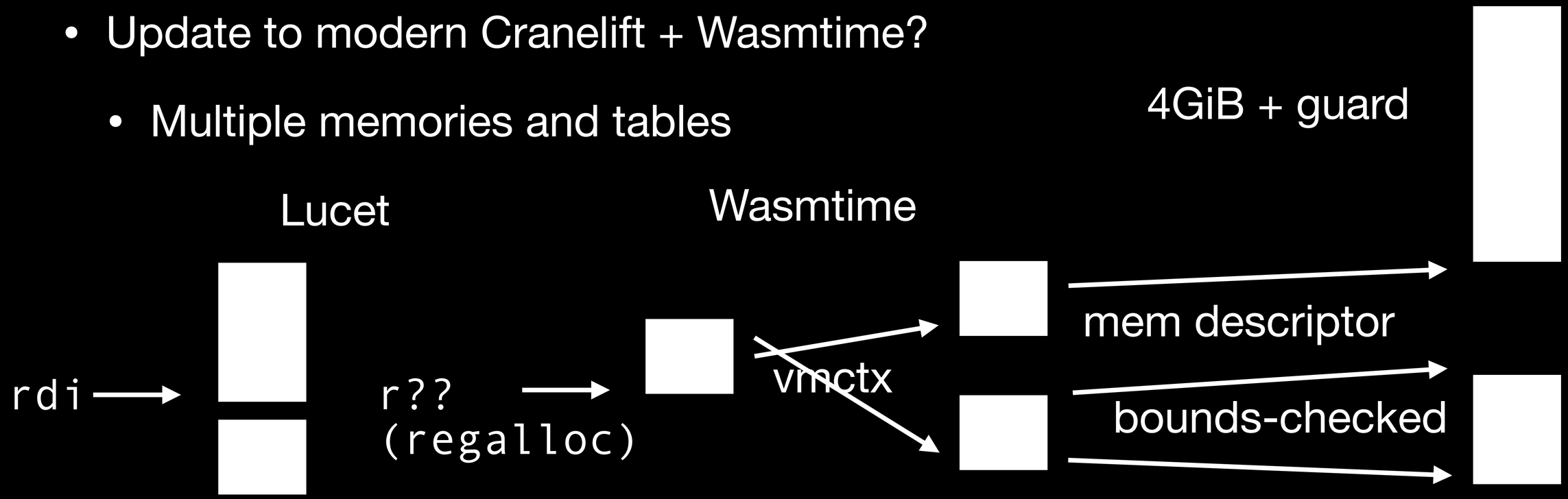
VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables



VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables



VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables — import vs. inline, dynamic vs. statlc, shared vs. non-shared, different guard region sizes, …

VeriWasm

- Verify memory safety (Wasm heap, funcref table), control-flow safety, and stack safety (accesses to stackframes) — focus here on Wasm heap
- Update to modern Cranelift + Wasmtime?
  - Multiple memories and tables — import vs. inline, dynamic vs. statlc, shared vs. non-shared, different guard region sizes, …
  - Also, the optimizer got better!

Wanted: the Perfect Verifier

- Linear-time and -space verification

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero’d reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero’d reg>
mov rdx, [r12]
```

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero’d reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero’d reg>
mov rdx, [r12]
```

Compute an i32

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero’d reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero’d reg>
mov rdx, [r12]
```

Compute an i32
Load from 4GiB-guard mem

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero’d reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero’d reg>
mov rdx, [r12]
```

Compute an i32

Load from 4GiB-guard mem

Bounds-check (Spectre)

Load from dynamic mem

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero'd reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero'd reg>
mov rdx, [r12]
```

Compute an i32

Load from 4GiB-guard mem

Bounds-check (Spectre)

Load from dynamic mem

How do we describe rax in the abstract domain??

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero'd reg>
mov rcx, [r10]
add r12, rax
cmp rax, r11
cmovae r12, <zero'd reg>
mov rdx, [r12]
```

Compute an i32

Load from 4GiB-guard mem

Bounds-check (Spectre)

Load from dynamic mem

How do we describe rax in the abstract domain??

rax < 4GiB && rax < r9 && rax < r11

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
                                  Compute an i32
                                  Load from 4GiB-guard mem
                                  Branch check (Spectre)
                                  dynamic mem
cmovae r12, <zero'd reg>
mov rdx, [r12]
```

Quadratic behavior!

How do we describe rax in the abstract domain??
rax < 4GiB && rax < r9 && rax < r11

Wanted: the Perfect Verifier

- Linear-time and -space verification

```
add eax, …
mov rbx, [r8+rax]
add r10, rax
cmp rax, r9
cmovae r10, <zero'd reg>
mov rcx, [r10]
```

Compute an i32

Load from 4GiB-guard mem

Bounds-check (Spectre)

Load from dynamic mem

Two separate parts combined later
-> Symbolic(123) and CompareResult(123, r9)??

How does this scale across GVN/value renames?

stract domain??

rax < 4GiB && rax < r9 && rax < r11

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
  - At least x86-64 and aarch64 (equally important production targets)

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
  - At least x86-64 and aarch64 (equally important production targets)
  - With most logic platform-independent

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
  - At least x86-64 and aarch64 (equally important production targets)
  - With most logic platform-independent
    - ISA-specific work should encode instruction semantics, but that’s it:

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
  - At least x86-64 and aarch64 (equally important production targets)
  - With most logic platform-independent
    - ISA-specific work should encode instruction semantics, but that’s it:

      mov rax, [r8 + 8*r9] -> rax = load(add(r8, scale(r9, 8)))

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
  - At least x86-64 and aarch64 (equally important production targets)
  - With most logic platform-independent
    - ISA-specific work should encode instruction semantics, but that’s it:

      mov rax, [r8 + 8*r9] -> rax = load(add(r8, scale(r9, 8)))

      ldr x20, [x19, w20, uxtw] -> x20 = load(add(x19, uextend(x20, 32, 64)))

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
  - Adding clever rewrites *might* require more domain knowledge encoded

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
  - Adding clever rewrites *might* require more domain knowledge encoded
  - … but we must not have to modify individual rules or passes to work with the verifier

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
- Prove safety of all memory loads+stores

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
- Prove safety of all memory loads+stores
  - Easily delineate our safety condition: “loads and stores occur according to some description/understanding of the runtime’s data layout”

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
- Prove safety of all memory loads+stores
  - Easily delineate our safety condition: “loads and stores occur according to some description/understanding of the runtime’s data layout”
    - This description is in the TCB; and the runtime (e.g. memory.grow) is; but the compiler is not

Wanted: the Perfect Verifier

- Linear-time and -space verification
- Portable across ISAs
- Easy to keep up-to-date as optimizer is modified
- Prove safety of all memory loads+stores
- Fast enough to run in production (translation validation on all compilations)

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
  - First: dynamic and static bounds, separately

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
  - First: dynamic and static bounds, separately -> nope, GVN can combine

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
  - First: dynamic and static bounds, separately -> nope, GVN can combine
  - Second: lattice that includes both kinds of bounds -> nope, multiple dynamic memories

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
  - First: dynamic and static bounds, separately -> nope, GVN can combine
  - Second: lattice that includes both kinds of bounds -> nope, multiple dynamic memories
  - Third: Set-of-upper-and-lower-bounds -> nope, not scalable

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
  - First: dynamic and static bounds, separately -> nope, GVN can combine
  - Second: lattice that includes both kinds of bounds -> nope, multiple dynamic memories
  - Third: Set-of-upper-and-lower-bounds -> nope, not scalable
  - Fourth: inequality solver (matrices + Gaussian reduction) -> nope, not scalable

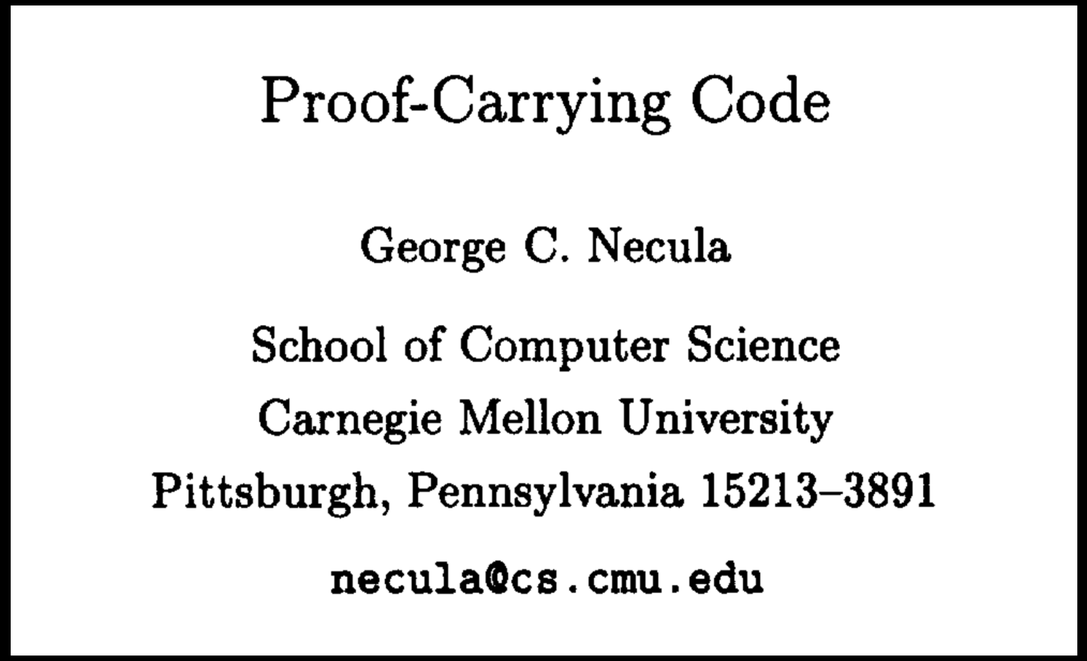
Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
- I think I have something that will work, with a trick

Spoiler: Work-in-Progress

- I’ve tried ~4 approaches; each time getting closer(?) on dynamic memories
  - What does work: static memories (like VeriWasm), over new Wasmtime data structures and Cranelift optimizations; 1% compile-time overhead
- I think I have something that will work, with a trick
- This is a workshop talk, after all!

Proof-Carrying Code?



Proof-Carrying Code

George C. Necula

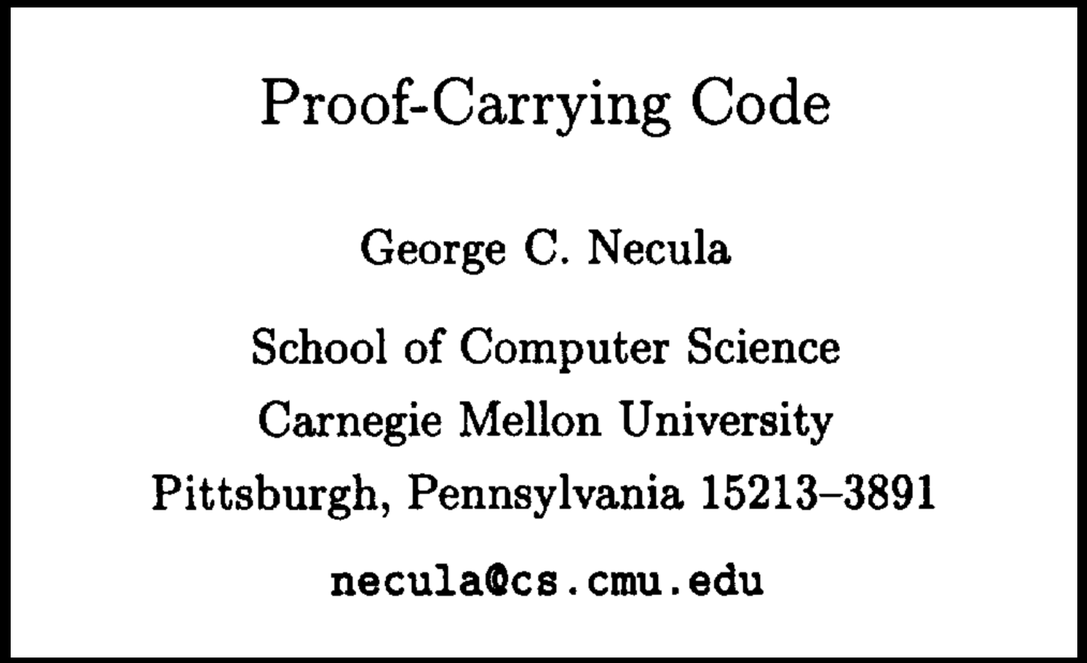
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

necula@cs.cmu.edu

POPL 1997

- Key idea: compiler emits proof steps to *check* — simpler than from-scratch *analysis* of binary artifact

Proof-Carrying Code?



Proof-Carrying Code

George C. Necula

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

necula@cs.cmu.edu

POPL 1997

- Key idea: compiler emits proof steps to *check* — simpler than from-scratch *analysis* of binary artifact
- Think of it like “typed assembly” + type-preserving compilation

Proof-Carrying Code

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80
    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

Proof-Carrying Code

Given fact: first arg is vmctx

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80

    mt0 = struct 88
            { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

Proof-Carrying Code

“memory types” describe layout

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80
    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

Proof-Carrying Code

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80

    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

facts on fields checked when loads are validated

Proof-Carrying Code

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80
    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

implicitly-validated fact based on range

Proof-Carrying Code

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80
    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000

block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

abstract-domain *add* operation

Proof-Carrying Code

```
function u0:0(i64 vmctx, i64) fast {
    gv3 ! mem(mt0, 0x0, 0x0) = vmctx
    gv4 ! mem(mt1, 0x0, 0x0) = load.i64 notrap aligned
                               readonly checked gv3+80

    mt0 = struct 88
          { 80: i64 readonly ! mem(mt1, 0x0, 0x0) }
    mt1 = memory 0x180000000


block7(v0: i64, v1: i64):
    v2 = ireduce.i32 v1
    v3 ! range(64, 0x0, 0xffffffff) = uextend.i64 v2
    v4 ! mem(mt1, 0x0, 0x0) = global_value.i64 gv4
    v5 ! mem(mt1, 0x0, 0xffffffff) = iadd v4, v3
    v6 = load.f64 little checked heap v5
}
```

checked load permitted only when offset in-bounds for memory type (here 4GiB)

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)              = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                 = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)            = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                  = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)               = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)       = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                          = load.i64 checked v8
    return v9
}
```

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)              = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                 = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)            = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                  = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)               = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)       = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                          = load.i64 checked v8
    return v9
}
```

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)             = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)           = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                 = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)              = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)      = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                         = load.i64 checked v8
    return v9
}
```

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)             = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)           = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                 = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)              = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)      = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                         = load.i64 checked v8
    return v9
}
```

- Too many pieces to put together: compare; symbolic addr; symbolic bound; select operator

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)             = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)           = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                 = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)              = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)      = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                         = load.i64 checked v8
    return v9
}
```

- Too many pieces to put together: compare; symbolic addr; symbolic bound; select operator
- Quadratic behavior arises from combination of these pieces when merged by optimizer

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1 ! dynamic_range(32, v1, v1): i32):
    v2 ! dynamic_range(64, v1, v1)             = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)                = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)           = global_value.i64 gv2
    v5 ! compare(uge, v1, gv2)                 = icmp.i64 uge v2, v4
    v6 ! dynamic_mem(mt1, v1, v1)              = iadd.i64 v3, v2
    v7 ! dynamic_mem(mt1, 0, 0, nullable)      = iconst.i64 0
    v8 ! dynamic_mem(mt1, 0, gv2-1, nullable) = select_spectre_guard v5, v7, v6
    v9                                         = load.i64 checked v8
    return v9
}
```

- Insight: if you can’t solve the problem, change the problem
  (carry through a “bounds-check” operator in the IR to a pseudo-machine-inst)

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1: i32):
    v2                                      = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)             = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)        = global_value.i64 gv2
    v5 ! dynamic_mem(mt1, 0, gv2-1, nullable) = dynamic_bound.i64 v3, v2, v4
    v6                                      = load.i64 checked v5
    return v6
}
```

- Insight: if you can’t solve the problem, change the problem
  (carry through a “bounds-check” operator in the IR to a pseudo-machine-inst)

Proof-Carrying Code: Dynamic Bounds

```
block0(v0 ! mem(mt0, 0, 0): i64, v1: i32):
    v2                                      = uextend.i64 v1
    v3 ! dynamic_mem(mt1, 0, 0)             = global_value.i64 gv1
    v4 ! dynamic_range(64, gv2, gv2)        = global_value.i64 gv2
    v5 ! dynamic_mem(mt1, 0, gv2-1, nullable) = dynamic_bound.i64 v3, v2, v4
    v6                                      = load.i64 checked v5
    return v6
}
```

- Insight: if you can’t solve the problem, change the problem (carry through a “bounds-check” operator in the IR to a pseudo-machine-inst)
- Subtle but important impact: separate value identity for property with separate validation status

Proof-Carrying Code: Dynamic Bounds

```
v2 = uextend.i64 v1

v3 = global_value.i64 gv1
v4 = global_value.i64 gv2

v5 = dynamic_bound.i64 v3, v2, v4



v6 = load.i64 checked v5
return v6
```

```
mov rax, …

mov rsi, [rdi+…]
mov rcx, [rdi+…]

xor r8, r8
add rsi, rax
cmp rax, rcx
cmovae rsi, r8 ;; zero if out-of-bounds

mov rax, [rsi]
```

- Emit `dynamic_bound` as “pseudoinstruction” (bundled machine instructions) and check as one unit: can show that combined semantics correspond

Symbolic Register Allocator Checker

- We’ve verified only up to virtual register code (VCode) — regalloc still in TCB
- Can we do translation validation on regalloc separately?

Symbolic Register Allocator Checker

```
add v0, v1
mov v3, [v2+v0*8]
mov [v4+v5], v3
ret
```

Register allocation: provide *abstraction* over real machine instructions with *virtual registers*

Symbolic Register Allocator Checker

```
add v0, v1
mov v3, [v2+v0*8]
mov [v4+v5], v3
ret
```

Equivalent?

```
add rax, rcx
mov r8, [r9+rax*8]
mov [r10+r11], r8
ret
```

Symbolic Register Allocator Checker

```
add v0, v1
mov v3, [v2+v0*8]
mov [v4+v5], v3
ret
```

Equivalent?

```
add rax, rcx
mov r8, [r9+rax*8]
mov [r10+r11], r8
ret
```

- Scan forward through code
- Track “contents” of each register
- Validate each arg gets expected vreg

Symbolic Register Allocator Checker

```
add v0, v1
mov v3, [v2+v0*8]
mov [v4+v5], v3
ret
```

Equivalent?

```
add rax, rcx
mov r8, [r9+rax*8]
mov [r10+r11], r8
ret
```

input: rax={v0}, rcx={v1}, r9={v2},
r10={v4}, r11={v5}
rax = {v0 (update)}
r8 = {v3}

- Scan forward through code
- Track “contents” of each register
- Validate each arg gets expected vreg

Symbolic Register Allocator Checker

- Aside: this is a fantastically effective way to write a new register allocator

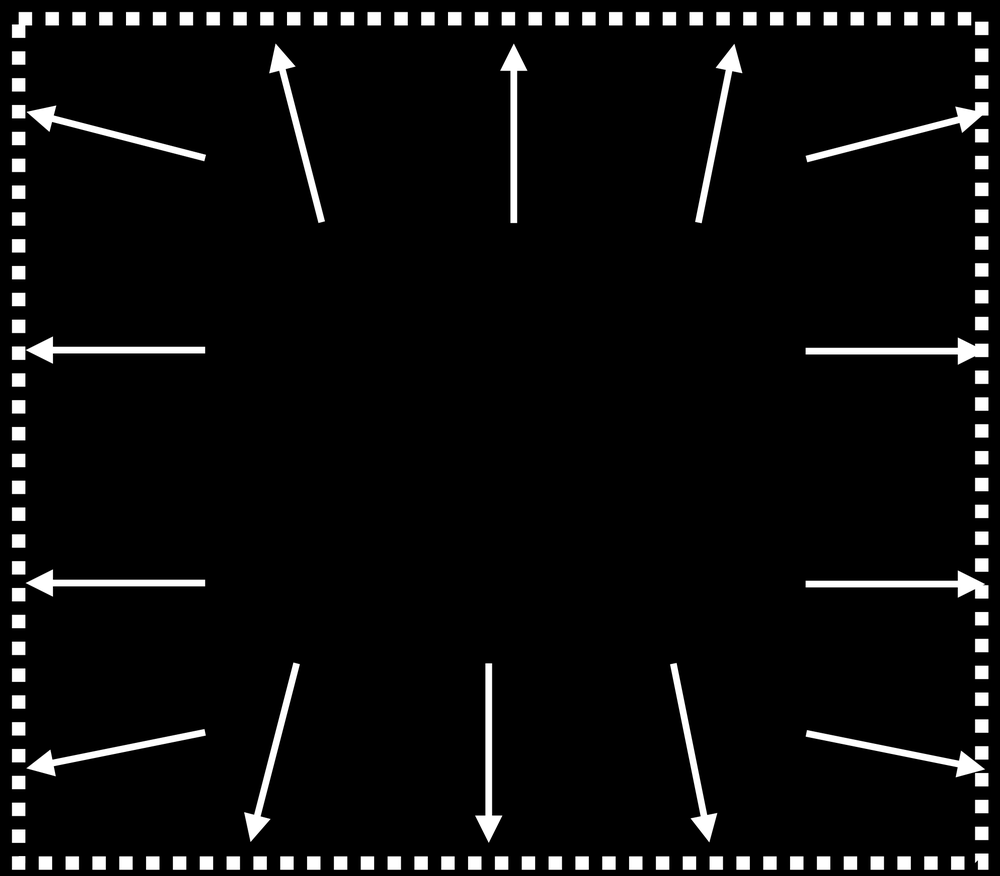
Symbolic Register Allocator Checker

- Aside: this is a fantastically effective way to write a new register allocator
  - Production regalloc often involves a lot of heuristics and edge-cases with funny constraints
  - I could not have found and resolved all edge-cases without it

Symbolic Register Allocator Checker

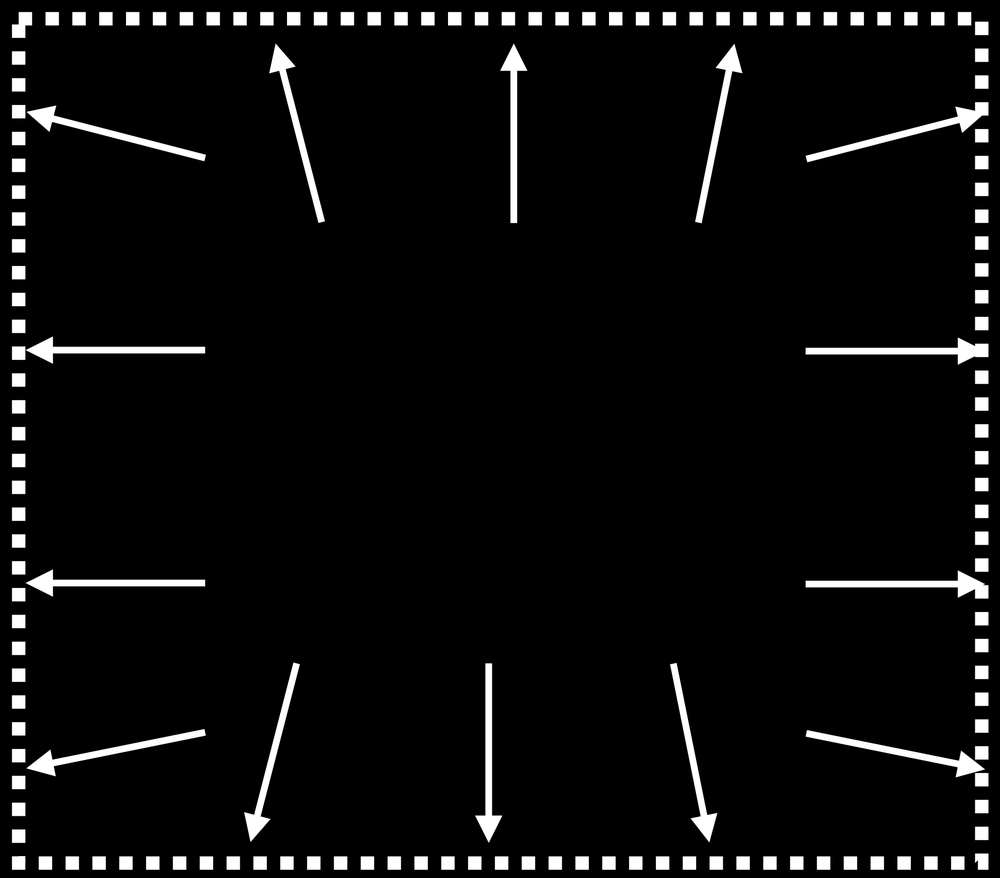
- Aside: this is a fantastically effective way to write a new register allocator
  - Production regalloc often involves a lot of heuristics and edge-cases with funny constraints
  - I could not have found and resolved all edge-cases without it
  - So effective that this is the *only* test method for `regalloc2` (no static suite)
    - We've never found a miscompilation due to RA in production in ~3 years

WebAssembly is Secure!



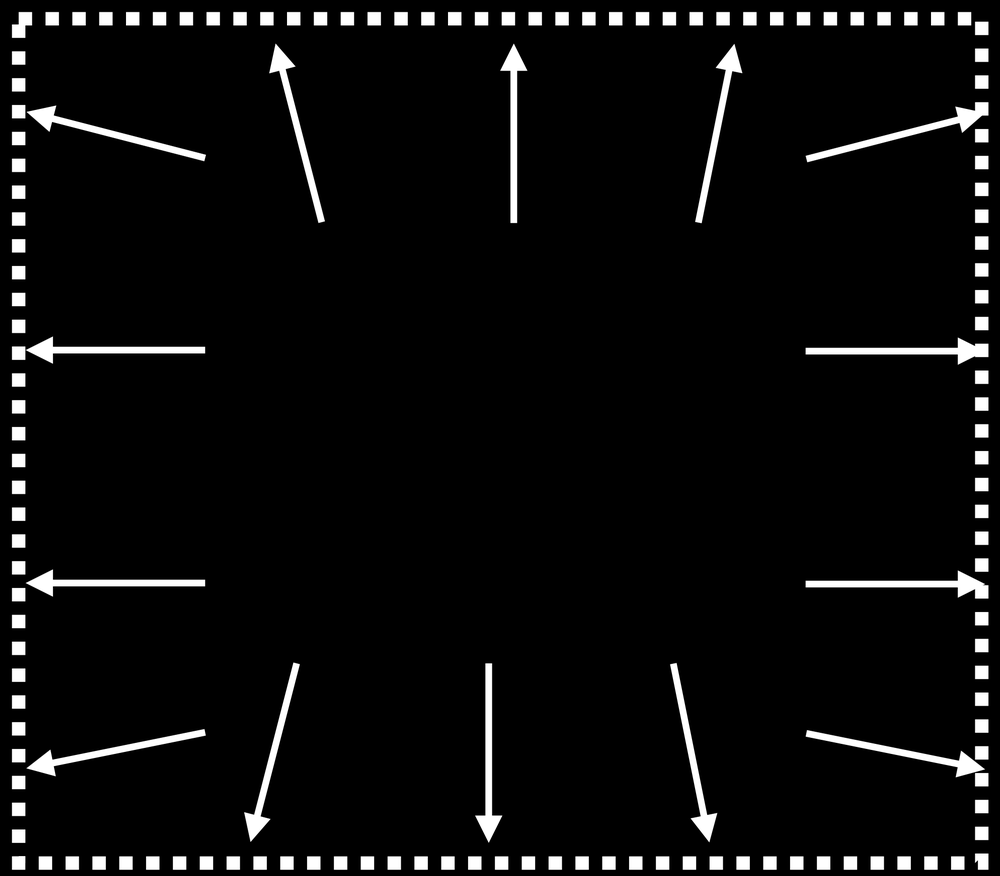
Sandbox boundary:

WebAssembly is Secure!



Sandbox boundary:
- Formal verification of instruction sel
- Translation validation of key parts
  (regalloc, memory sandboxing?)

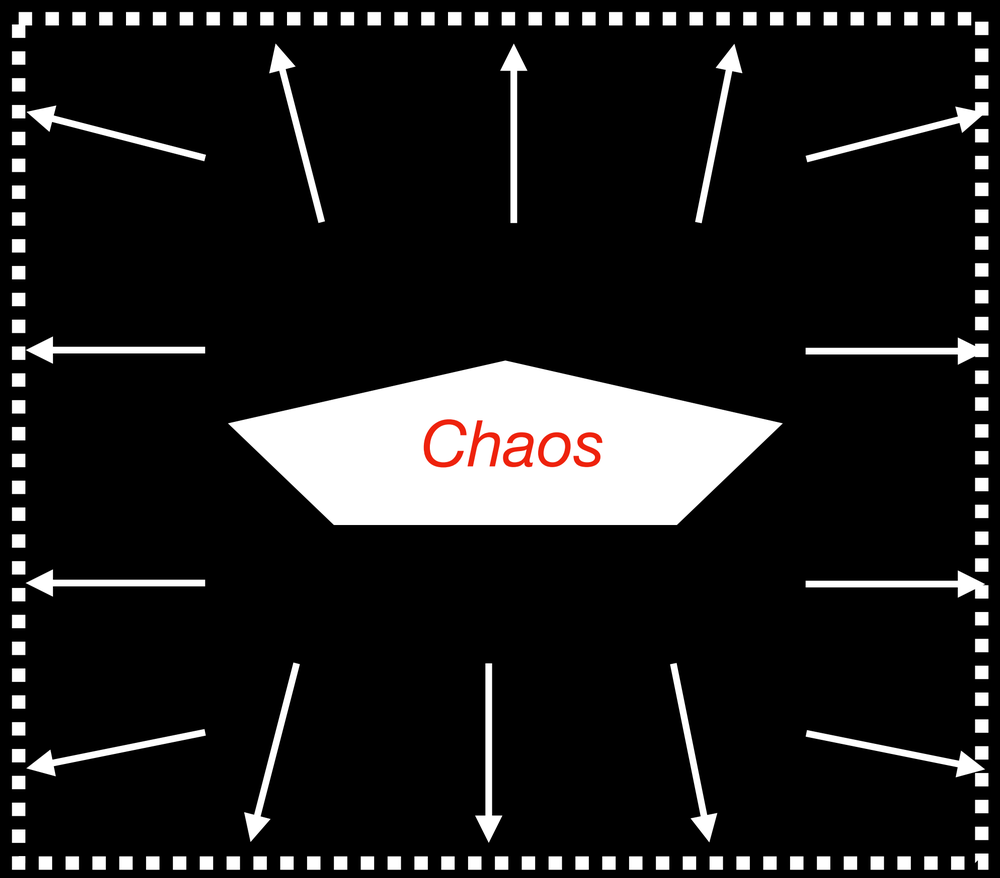
WebAssembly is Secure!



Sandbox boundary:
- Formal verification of instruction sel
- Translation validation of key parts
  (regalloc, memory sandboxing?)

✅ *Secure*

…WebAssembly is Secure?



Sandbox boundary:
- Formal verification of instruction sel
- Translation validation of key parts
  (regalloc, memory sandboxing?)

?? Secure?

WebAssembly is Secure?

- What is the *threat model*?

WebAssembly is Secure?

- What is the *threat model*?
  - Code attempting to exceed permissions on *system*: OK!

WebAssembly is Secure?

- What is the *threat model*?
  - Code attempting to exceed permissions on *system*: OK!
  - Code attempting to exceed permissions *inside guest*

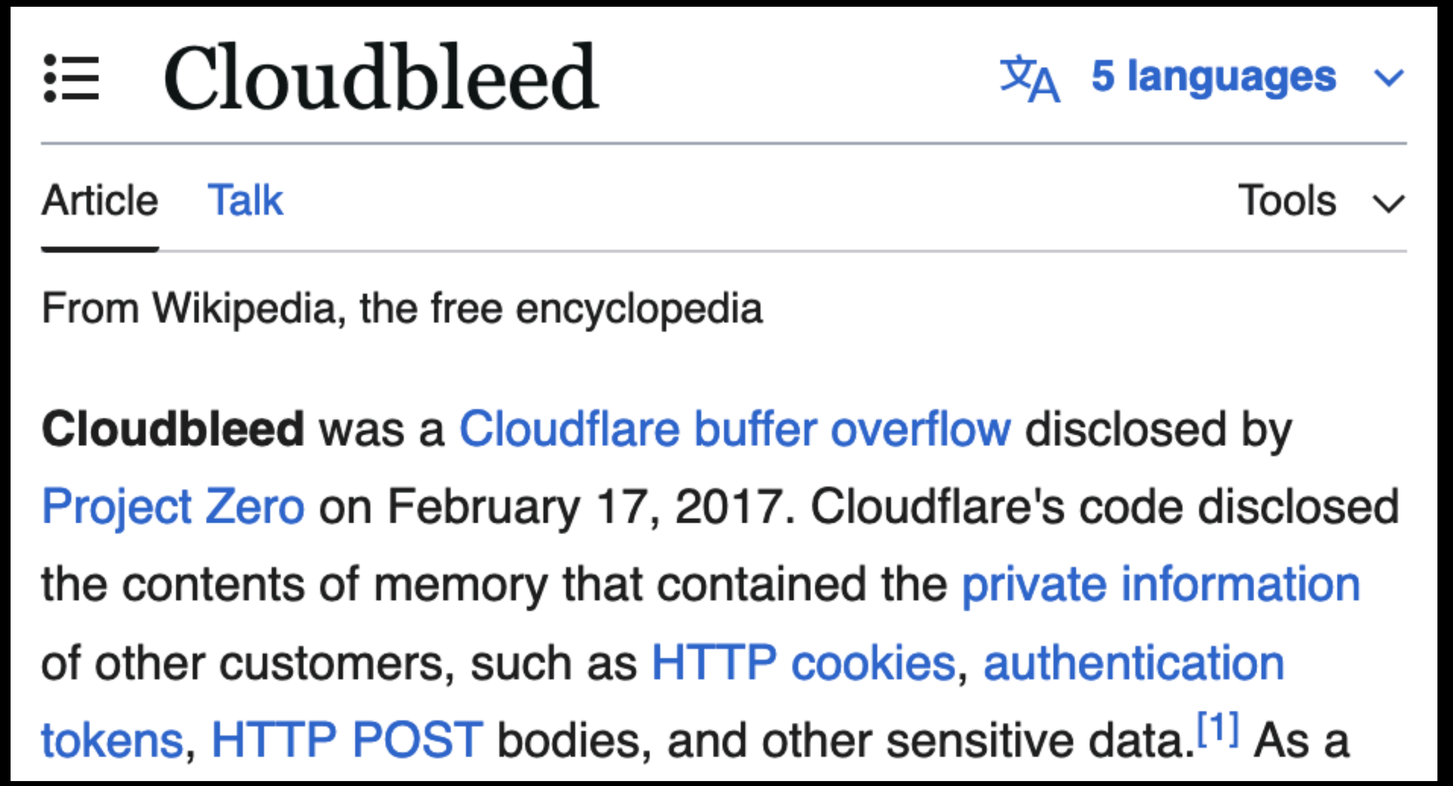
WebAssembly is Secure?

- What is the *threat model*?
  - Code attempting to exceed permissions on *system*: OK!
  - Code attempting to exceed permissions *inside guest*
    - Exploit runtime / language implementation bugs to…
      - …Observe other requests' data

WebAssembly is Secure?

- What is the *threat model*?
  - Code attempting to exceed permissions on *system*: OK!
  - Code attempting to exceed permissions *inside guest*
    - Exploit runtime / language implementation bugs to…
      - …Observe other requests' data
      - …Subvert authorization logic
      - …Inject malicious content
- We still need a *correct language implementation* for *application security*

WebAssembly is Secure?



Cloudbleed

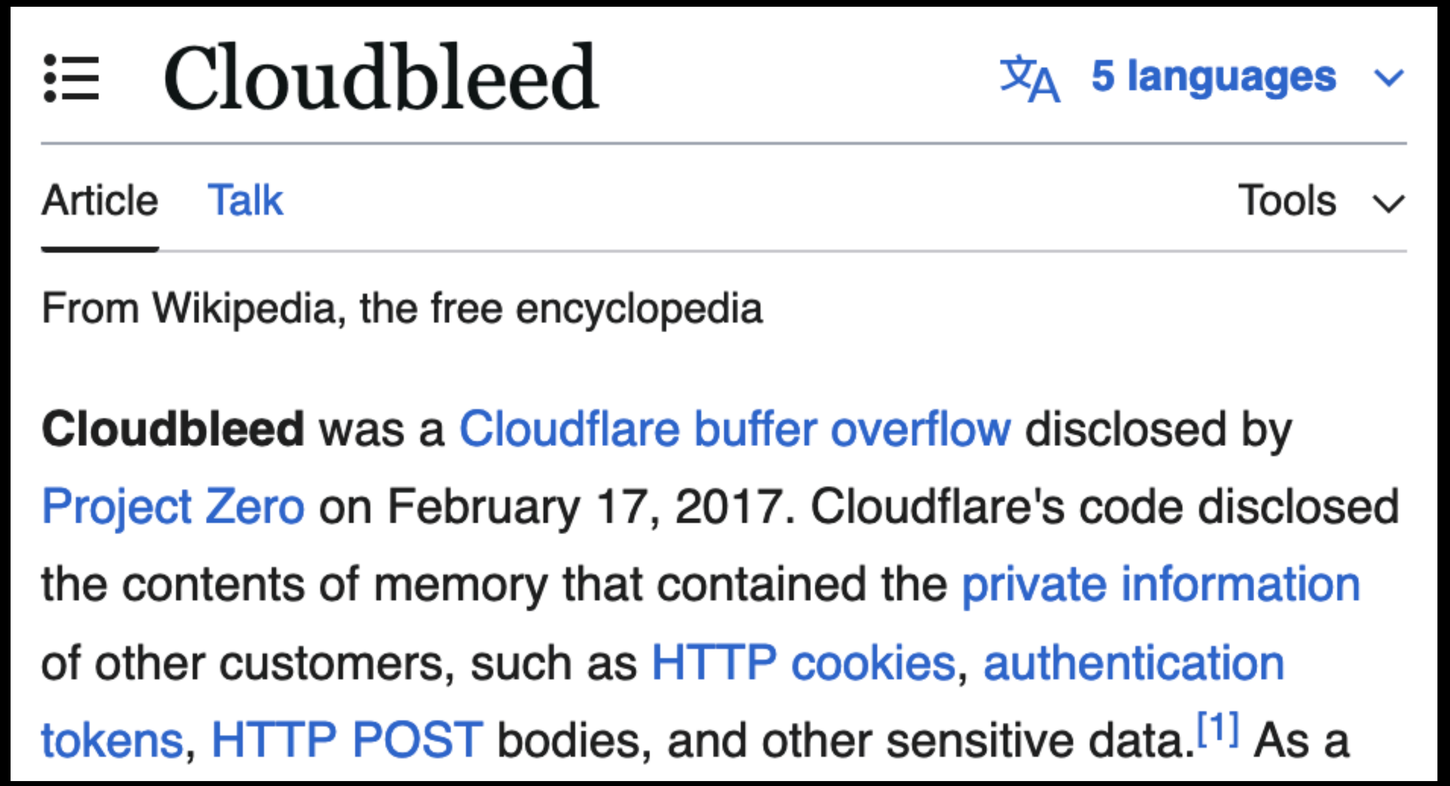
5 languages

Article Talk Tools

From Wikipedia, the free encyclopedia

Cloudbleed was a Cloudflare buffer overflow disclosed by Project Zero on February 17, 2017. Cloudflare's code disclosed the contents of memory that contained the private information of other customers, such as HTTP cookies, authentication tokens, HTTP POST bodies, and other sensitive data.[1] As a

WebAssembly is Secure?



Cloudbleed

5 languages

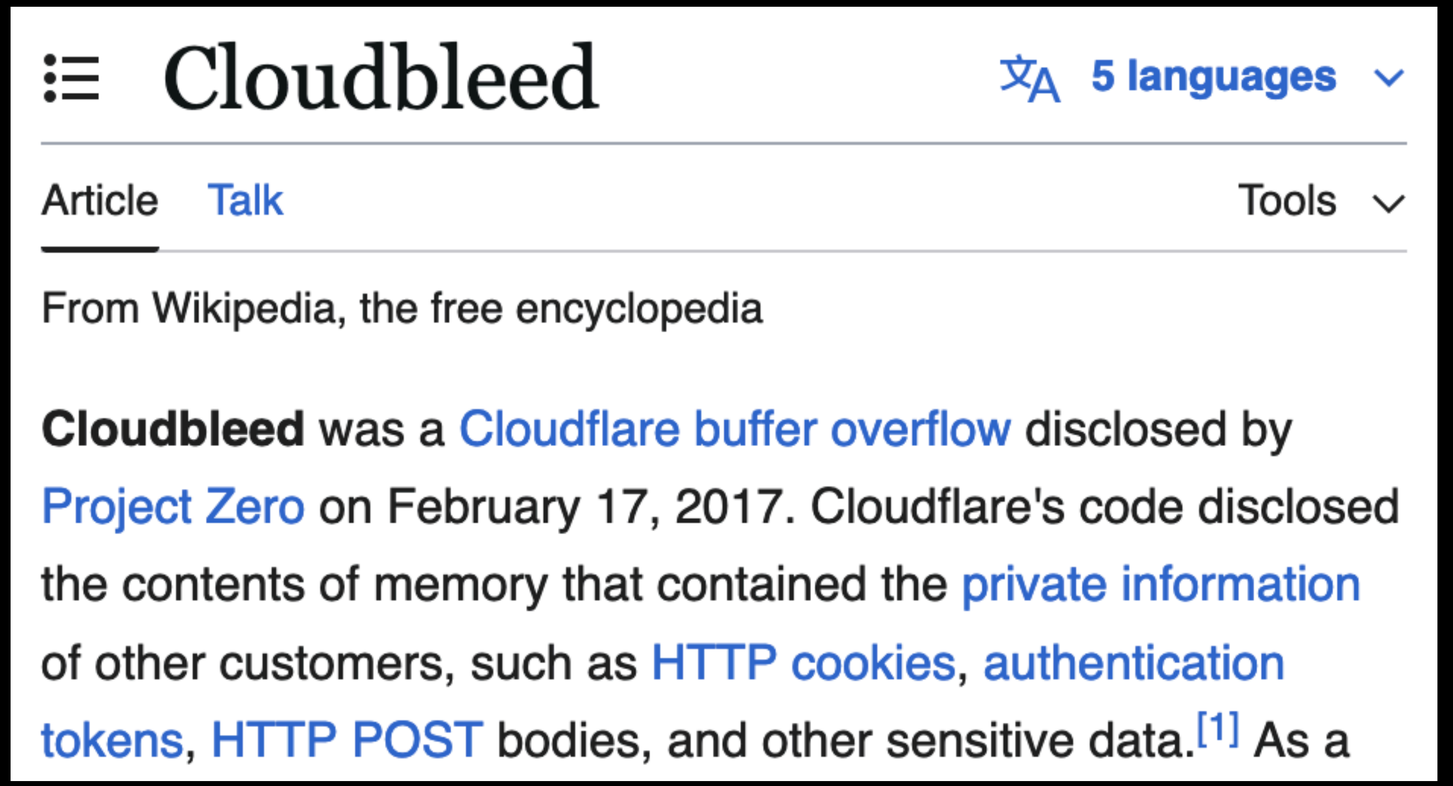
Article Talk Tools

From Wikipedia, the free encyclopedia

Cloudbleed was a Cloudflare buffer overflow disclosed by Project Zero on February 17, 2017. Cloudflare's code disclosed the contents of memory that contained the private information of other customers, such as HTTP cookies, authentication tokens, HTTP POST bodies, and other sensitive data.[1] As a

Defense-in-depth: *per-request isolation*
-> even a buggy runtime cannot allow cross-user leakage

WebAssembly is Secure?



Cloudbleed

5 languages

Article Talk Tools

From Wikipedia, the free encyclopedia

Cloudbleed was a Cloudflare buffer overflow disclosed by Project Zero on February 17, 2017. Cloudflare's code disclosed the contents of memory that contained the private information of other customers, such as HTTP cookies, authentication tokens, HTTP POST bodies, and other sensitive data.[1] As a

Defense-in-depth: *per-request isolation*

-> even a buggy runtime cannot allow cross-user leakage

AKA: put the Wasm sandbox boundary *between requests*

Building a Correct(-ish) JavaScript Runtime

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else’s runtime

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime
  - Optimization logic bugs (especially type confusion) account for *many* CVEs

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime
  - Optimization logic bugs (especially type confusion) account for *many* CVEs
  - But don't you want performance?! (yes… we'll get there)

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime
  - Optimization logic bugs (especially type confusion) account for *many* CVEs
  - But don't you want performance?! (yes… we'll get there)
  - … and it's all we can run on Wasm anyway, today

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime
  - Optimization logic bugs (especially type confusion) account for *many* CVEs
  - But don't you want performance?! (yes… we'll get there)
  - … and it's all we can run on Wasm anyway, today
- Step 3: … fix performance

Building a Correct(-ish) JavaScript Runtime

- Step 1: use someone else's runtime
  - Our friends at Mozilla do lots of fuzzing, testing, security things; SpiderMonkey is battle-tested
- Step 2: avoid the JIT compiler of that runtime
  - Optimization logic bugs (especially type confusion) account for *many* CVEs
  - But don't you want performance?! (yes… we'll get there)
  - … and it's all we can run on Wasm anyway, today
- Step 3: … fix performance (???)

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language

An Ode to Interpreters

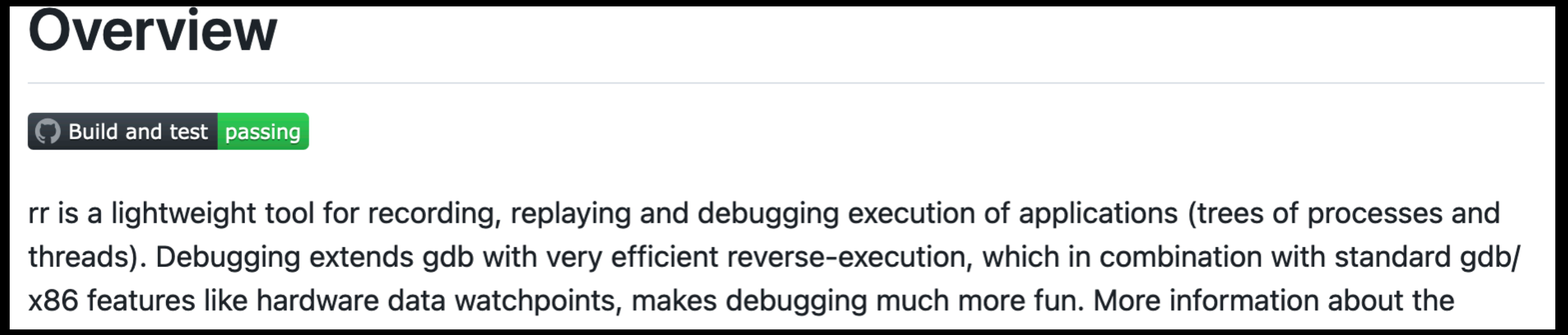
- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

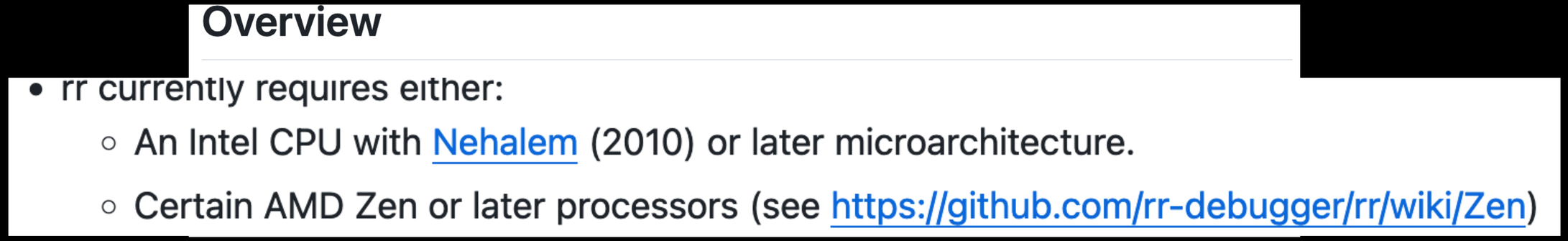
An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms



An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms



An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

Native x86_64-linux

rr

perf

gdb

valgrind

asan

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

Native x86_64-linux

rr

perf

gdb

valgrind

asan

Wasmtime

gdb kind of works

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

Native x86_64-linux

rr

perf

gdb

valgrind

asan

Wasmtime

gdb kind of works

but steps through runtime too

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

Native x86_64-linux

rr

perf

gdb

valgrind

asan

Wasmtime

gdb kind of works

but steps through runtime too

DWARF transform assertion failures

An Ode to Interpreters

- An interpreter is the *easiest* — and thus *most likely to be correctly written* — implementation of a language
- Typical tiering architectures in JITs mean that interpreters are “simple” — focus on correctness rather than (too much) performance
- An interpreter is *portable* — and thus can be developed on native platforms

Native x86_64-linux

rr

perf

gdb

valgrind

asan

Wasmtime

gdb kind of works

but steps through runtime too

DWARF transform assertion failures

Single Source of Truth

- Interpreter will exist anyway (JIT tiers, fallback); is easier to get right; works fine on Wasm (naturally portable); it's just… slow
- Can we keep the interpreter as the *only* language implementation, and somehow derive a compiler from it?

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

func:

ADD
RET

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

func:

ADD
RET

```
func() {
  auto a = pop();
  auto b = pop();
  push(a + b);
  return pop();
}
```

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

func:

ADD
RET

```
func() {
  auto a = pop();
  auto b = pop();
  push(a + b);
  return pop();
}
```

Key insight: Wasm is a small, introspectable, well-behaved IR;
partial evaluation should be tractable (moreso than on native code)

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)
  - Very very very important guiding principle: *no magic*, only semantics-preserving transforms; specialized function behaves identically to original

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)
  - Very very very important guiding principle: *no magic*, only semantics-preserving transforms; specialized function behaves identically to original
  - Gives us a compiler “for free” once we have an interpreter

Specialization Intrinsics

```
void interp(bytecode* pc) {

  while (true) {
    switch (*pc++) {
      case OP1:
        …

        break;
      case OP2:
        …

        break;
    }
  }
}
```

Specialization Intrinsics

```
void interp(bytecode* pc) {
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        break;
      case OP2:
        …
        break;
    }
  }
}
```

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        weval::update_context(pc);
        break;
      case OP2:
        …
        weval::update_context(pc);
        break;
    }
  }
}
```

Specialization Intrinsics

1. “No magic”: only expand code where interpreter specifies via *context* mechanism
2. Partially evaluate iterations of the interpreter loop in a *context-sensitive* way, where the context is the bytecode PC
3. … and that’s it.

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        weval::update_context(pc);
        break;
      case OP2:
        …
        weval::update_context(pc);
        break;
    }
  }
}
```

Discussion: Compilers from Interpreters

- This works; shipping in StarlingMonkey runtime; ~2-3x speedups

Discussion: Compilers from Interpreters

- This works; shipping in StarlingMonkey runtime; ~2-3x speedups
- We are *deriving a JIT from first principles* from an interpreter

Discussion: Compilers from Interpreters

- This works; shipping in StarlingMonkey runtime; ~2-3x speedups
- We are *deriving a JIT from first principles* from an interpreter
- We are avoiding doing anything special or language/JIT-engine-specific

Discussion: Compilers from Interpreters

- This works; shipping in StarlingMonkey runtime; ~2-3x speedups
- We are *deriving a JIT from first principles* from an interpreter
- We are avoiding doing anything special or language/JIT-engine-specific
- We think we can get more optimizations by writing *semantics-preserving rules*
  - E.g., profile-guided speculative inlining + box-unbox elision to get type-specialized unboxing in JS

Discussion: Correctness-Focused Runtimes

- Correct software is a never-fully-attained goal (realistically)
- But we can carefully delineate abstraction boundaries and validate them separately

Discussion: Correctness-Focused Runtimes

- Observation: limited formal methods can be practical in practice
  - SMT-based checking of compiler lowering rules
  - Symbolic checker of register allocator
  - Maybe? proof-carrying code for sandboxing logic

Discussion: Correctness-Focused Runtimes

- Observation: limited formal methods can be practical in practice
  - SMT-based checking of compiler lowering rules
  - Symbolic checker of register allocator
  - Maybe? proof-carrying code for sandboxing logic
- Observation: meta-compilers (deriving compilers from simpler representations) can be practical in practice
  - weval is much smaller than the full compiler-to-Wasm would have been

Discussion: Correctness-Focused Runtimes

- Observation: limited formal methods can be practical in practice
  - SMT-based checking of compiler lowering rules
  - Symbolic checker of register allocator
  - Maybe? proof-carrying code for sandboxing logic
- Observation: meta-compilers (deriving compilers from simpler representations) can be practical in practice
  - weval is much smaller than the full compiler-to-Wasm would have been
- Wasm has set an excellent precedent for explicit semantics, static typing, and focus on small clean abstractions

Thanks! Questions?