Chihuahua: A WASM sandbox

Derek Miller, Dominic Mulligan, Shale Xiong, Hugo Vincent

Arm Research

Berkeley WASM outside the browser workshop

9 August 2019

Chihuahua motivation

Part of a wider project ("*Veracruz*") looking at secure processing environments (SPEs)

Goal: sandbox for running small loadable programs inside SPEs

- Some issues with measurement/executable pages on some SPEs
- Provide confidentiality guarantees for loaded program
- Protection of the host machine from loaded program (e.g. Rowhammer)
- Ability to mitigate µarchitectural vulnerabilities without forcing program recompilation
- Enforce dynamic policies on code running in sandbox
- High-level of trust in the sandbox

Some security engineering principles

Adopt some design principles when developing Veracruz:

- All code should be easily auditable by humans
- Every line of code in the SPE should have a purpose
- Any code not strictly required to be in the SPE should be moved outside

Most code running in SPE is written in Rust. Exception: Chihuahua which is written in C…

- Static analysis used to obtain memory safety, rather than property of language
- Also want to verify properties of Chihuahua

C is (perversely!) one of the best languages to use for verification: it's unsafe, everyone knows it's unsafe, and it's everywhere: so lots of tools exist

Chihuahua, concretely

Chihuahua is a “hardware-like” VM (in JVM sense) implementing Zocalo ISA:

- Half-way house between WASM stack machine and RISC load/store architecture
- Fixed-width instruction encoding (32-bits), load-store architecture
- Separate integer, floating point GPRs in both 32- and 64-bit variants

Most opcodes and semantics borrowed from WASM:

- WASM has structured control-flow, Zocalo de-structures this using jumps and condition flags
- Interpreter main loop simpler to audit and verify (just fetch/execute, no control flow)

Zocalo Virtual Machine:

- Harvard Architecture: separate code, main memories, only fetch reads from code memory
- Parameters passed via operand stack, only `push` and `pop` manipulate this
- Separate return address stack: only `call` and `return` manipulate this
- Deterministic: no undefined behaviours, meaning anything strange causes immediate program termination

Why this approach?

WASM is designed for JITing: this is how most (all?) browsers use WASM

- JITs hard to audit, especially sophisticated ones: as line-count increases probability of memory errors, security holes increases. Chihuahua is simple…
- Some SPEs don't sit neatly with JITs
- Dynamic policies easier to enforce with Chihuahua, e.g. resource counts, trace properties of program
- Potentially want to verify WASM to Zocalo translation making use of e.g. Conrad Watt's work in Isabelle/HOL

But…

- We leave potentially a lot of performance on the table
- One idea: we could try writing a high-assurance WASM JIT (interest in collaborating?)

Why WASM in the middle?

Many compilers targeting WASM: use it as an IR and a path from C to Zocalo

Translation tool, `wasm2zocalo`, reuses reference interpreter written in OCaml

- Reuses front-end, parses WASM binary and produces OCaml AST of the WASM program
- Walk the AST translating into Zocalo instructions
- Separate assembler that does some peephole optimisations, generates addresses etc
- Translated 85% of WASM instruction set

Things to be done:

- Control-flow destructuring,
- Setup of subprocedure environment prior to function call,
- Hoist large immediate constant loads into memory loads,
- Add explicit pushes and pops
- Convert 32-bit addresses to 64-bit addresses

arm

Zocalo implementation

Zocalo is implemented in pure C, around 6,000 lines of heavily-commented code

- Merge requests code-reviewed by another team member,
- Compiled with `clang` with all warnings and errors turned on

Use **C Bounded Model Checker** (CBMC) on Zocalo implementation:

- Capable of verifying properties of C code (pre/post-conditions, asserts etc)
- Also detecting undefined behaviours, code-smells in C (e.g. overflow, memory/indexing errors)

Also run Frama-C, a static-analysis/verification platform on code

- Disadvantage: requires extensive EACSL annotations embedded in code
- Mostly does what CBMC does, but more complex to use and understand, so capability a little bitrotted

Other potential avenues: TIS interpreter and Cerberus C tools from Peter Sewell's group

- Arm is a TIS licensee…

What properties do we verify?

Around 10,000 lines of C code for CBMC verification testbenches:

- Verify that encode and decode are correct (mutual partial inverses)
- Verify Zocalo semantics is as expected † Floating-point instructions cause CBMC to die ☹
- Verify machine invariants are preserved by execution: e.g. PC always valid
- Memory, stack read and write functions have correct properties
- Unreachable code paths are marked, unreachability checked for valid inputs
- All Zocalo instructions in execute, decode functions are handled using cover properties

Takes 72 hours to run full test bench:

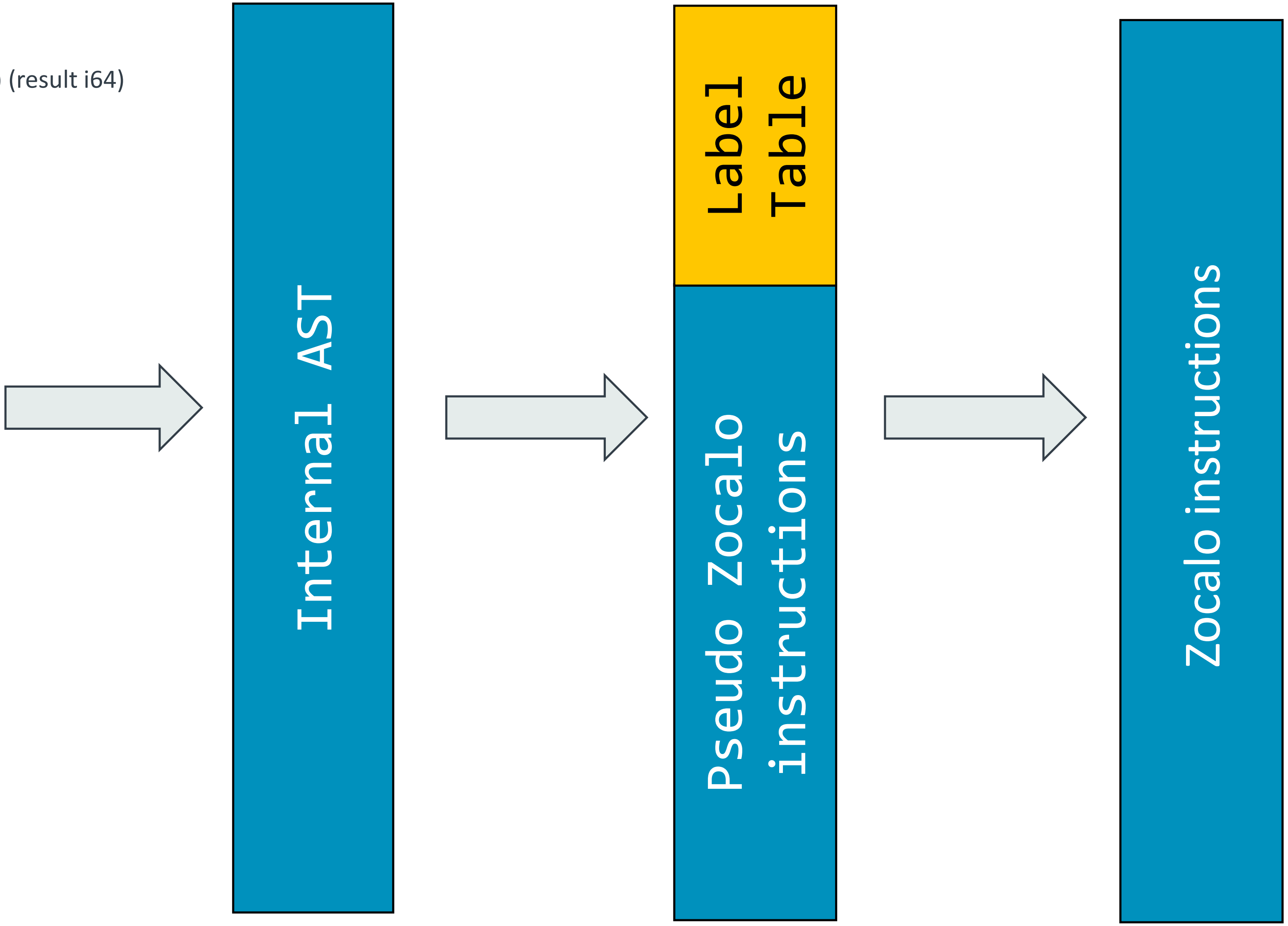
  - 90% of time spent verifying execution
  - Flushed out lots of bugs

arm

Translating WASM bytecode to Zocalo

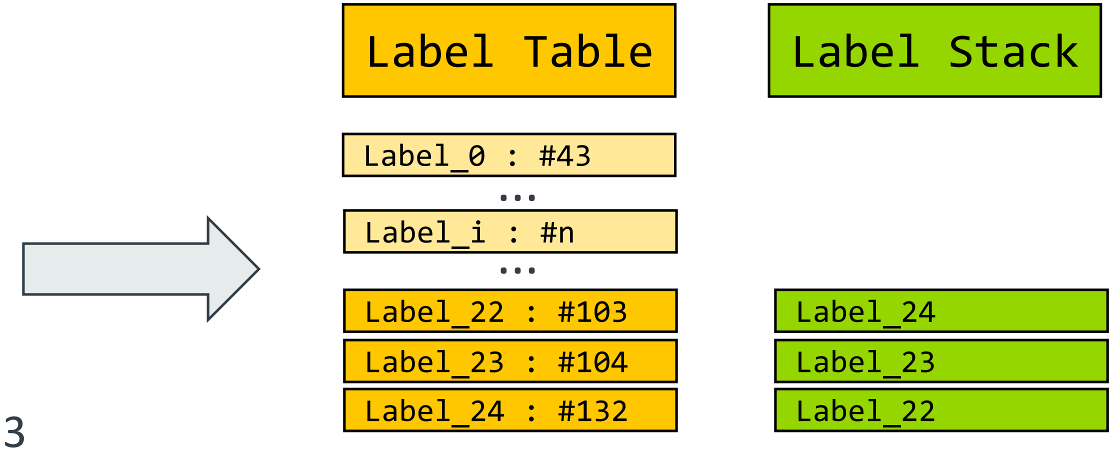
WASM to Zocalo overview

```
(func (;0;) (type 0) (param i32 i64) (result i64)
  (local i32 i32 i64 i64)
  local.get 1
  i64.const 0xffffffff
  i64.ne
  if  ;; label = @1
    loop  ;; label = @2
      ……
      loop  ;; label = @3
        local.get 2
        ……
        i64.lt_u
        br_if 1 (;@2;)
      end
      call 1
      ……
    end
  end
  local.get 4)
(func (;1;) …….)
```



WASM to Zocalo

```
(func (;0;) (type 0) (param i32 i64) (result i64)
  (local i32 i32 i64 i64)
#100  local.get 1
#101 i64.const 0xffffffff
#102  i64.ne
#103 if  ;; label = @1
#104 loop  ;; label = @2
    ……
    #132 loop  ;; label = @3
     local.get 2
     ……
     i64.lt_u
     br_if 1 (;@2;)
    end
    call 1
    ……
   end
  end
  local.get 4)
 (func (;1;) .........)
```



WASM to Zocalo

```
(func (;0;) (type 0) (param i32 i64) (result i64)
  (local i32 i32 i64 i64)
#100  local.get 1
#101 i64.const 0xffffffff
#102  i64.ne
#103 if  ;; label = @1
#104 loop  ;; label = @2
    ......
    #132 loop  ;; label = @3
      local.get 2
      ......
      i64.lt_u
      br_if 1 (;@2;)
    end
    call 1
    ......
   end
  end
  local.get 4)
(func (;1;) ........)
```

Label Table

Label_0 : #43
...
Label_i : #n
...
Label_22 : #103
Label_23 : #104
Label_24 : #132

Label Stack

Label_24
Label_23
Label_22

Pseudo Zocalo instructions

br_if label_23

WASM to Zocalo

R31 points to parameters and local variables stack. Initially it pointed to end of heap.

```
(func (;0;) (type 0) (param i32 i64) (result i64)
(local i32 i32 i64 i64)
#100  local.get 1
#101 i64.const 0xffffffff
#102  i64.ne
#103 if  ;; label = @1
#104 loop  ;; label = @2
   ......
   #132 loop  ;; label = @3
    local.get 2
    ......
    i64.lt_u
    br_if 1 (;@2;)
   end
   call 1
   ......
  end
 end
 local.get 4)
#323 (func (;1;) ........)
```

Label Table

```
Label_0 : #43
...
Label_i : #n
...
Label_12 : #323
...
Label_k : #m
```

Pseudo Zocalo instructions

```
// reserve space for local variables
sub r31 r31 0x1c
// pop and store parameters
pop r0
sub r31 r31 0x4
store r0 [r31+0x0]
…
// call the function
call label_12
```