WasmBounds

Eliminating Memory Bounds Checks with Abstract Interpretation

Emma Sudo

Stanford

Why do we need a tool to eliminate memory bounds checks?

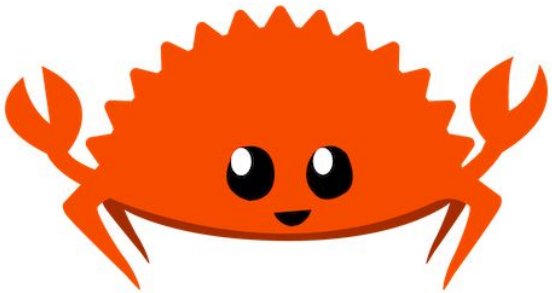
In Wasm 1.0, OOB accesses are efficiently caught using virtual memory/guard pages.

With 64-bit Memory, Single-Byte Page Size, and Embedded Systems, software bounds checking is often used at a cost.

```
if base + offset > length of memory:
   trap()
```

Wait! We already know that some accesses are safe.

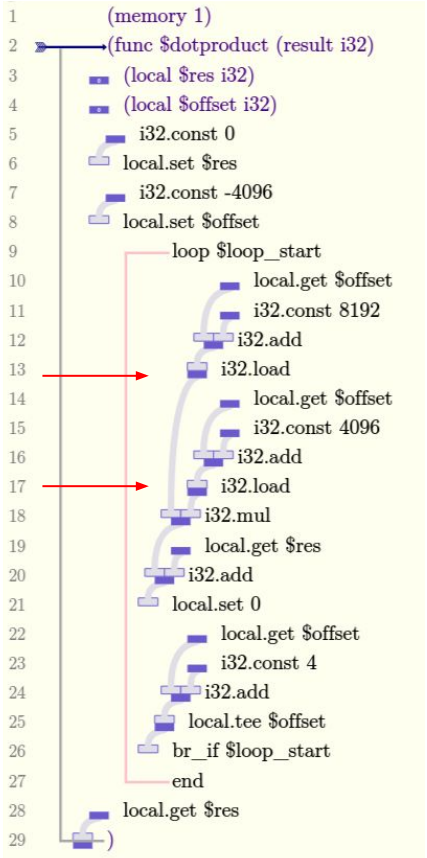
For example, safe languages are already bounds checking accesses to container types.



It’s clear at the source level that the accesses are safe…

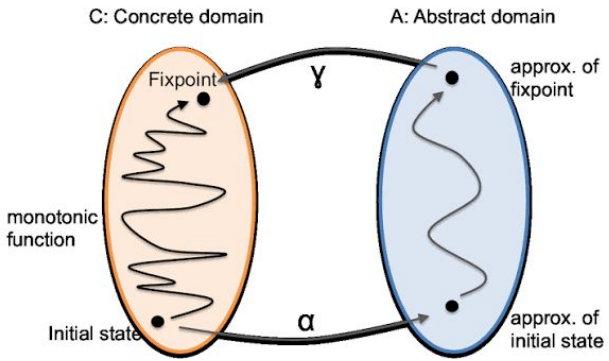
```
const int N = 1024;

int first[N];
int second[N];

int get_first(int n) {
    return first[n % N];
}

int get_second(int n) {
    return second[n % N];
}

int dotproduct(void) {
  int ret = 0;
  for (unsigned int i = 0; i < N; ++i) {
    ret += get_first(i) * get_second(i);
  }
  return ret;
}
```

…but it’s less obvious at the Wasm binary level



We can use abstract interpretation to over-approximate the addresses and eliminate the associated bounds check for provably safe accesses.

1. Translate each Wasm instruction to operate over an abstract domain (strided, sign-agnostic integer intervals)
2. Run through the program using the abstract interpreter multiple times until the abstract store reaches a fix point.



{0, 4, 8, 12} → Interval { lower: 0, upper: 12, stride: 4}

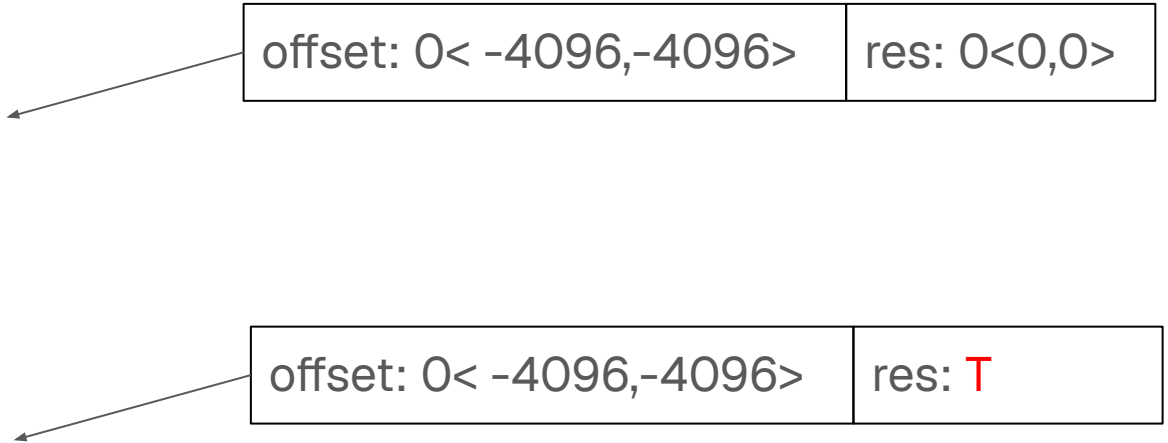
4<0, 12>

Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9         loop $loop_start
10              local.get $offset
11              i32.const 8192
12            i32.add
13            i32.load
14              local.get $offset
15              i32.const 4096
16            i32.add
17            i32.load
18          i32.mul
19            local.get $res
20        i32.add
21        local.set 0
22              local.get $offset
23              i32.const 4
24          i32.add
25          local.tee $offset
26        br_if $loop_start
27        end
28    local.get $res
29  )
```

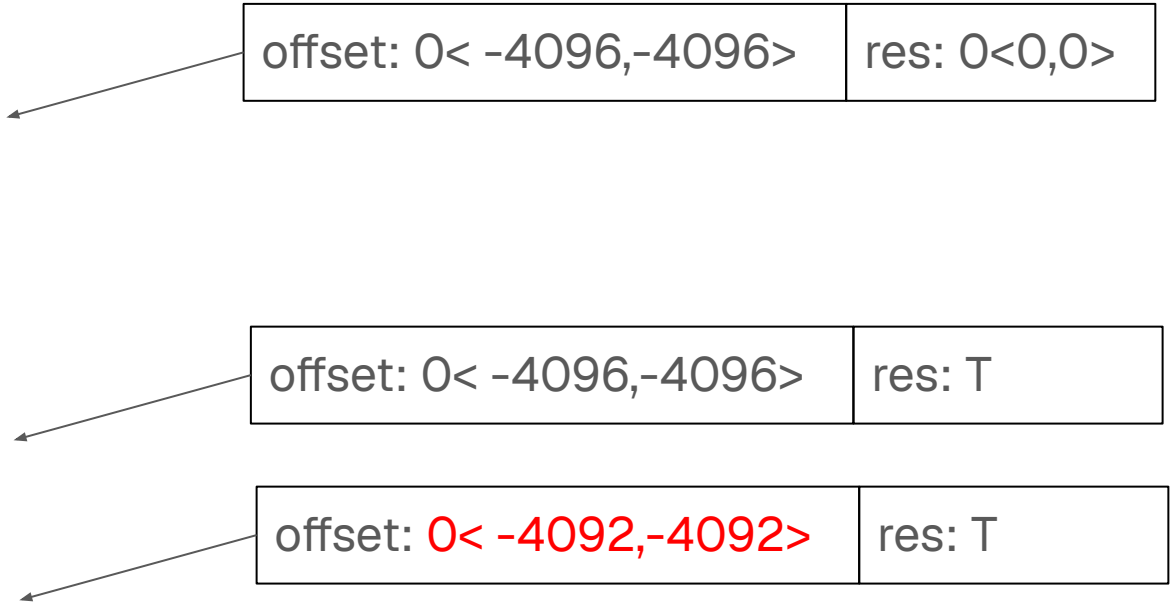
Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9         loop $loop_start
10              local.get $offset
11              i32.const 8192
12            i32.add
13            i32.load
14              local.get $offset
15              i32.const 4096
16            i32.add
17            i32.load
18          i32.mul
19            local.get $res
20        i32.add
21        local.set 0
22              local.get $offset
23              i32.const 4
24            i32.add
25          local.tee $offset
26        br_if $loop_start
27        end
28    local.get $res
29  )
```



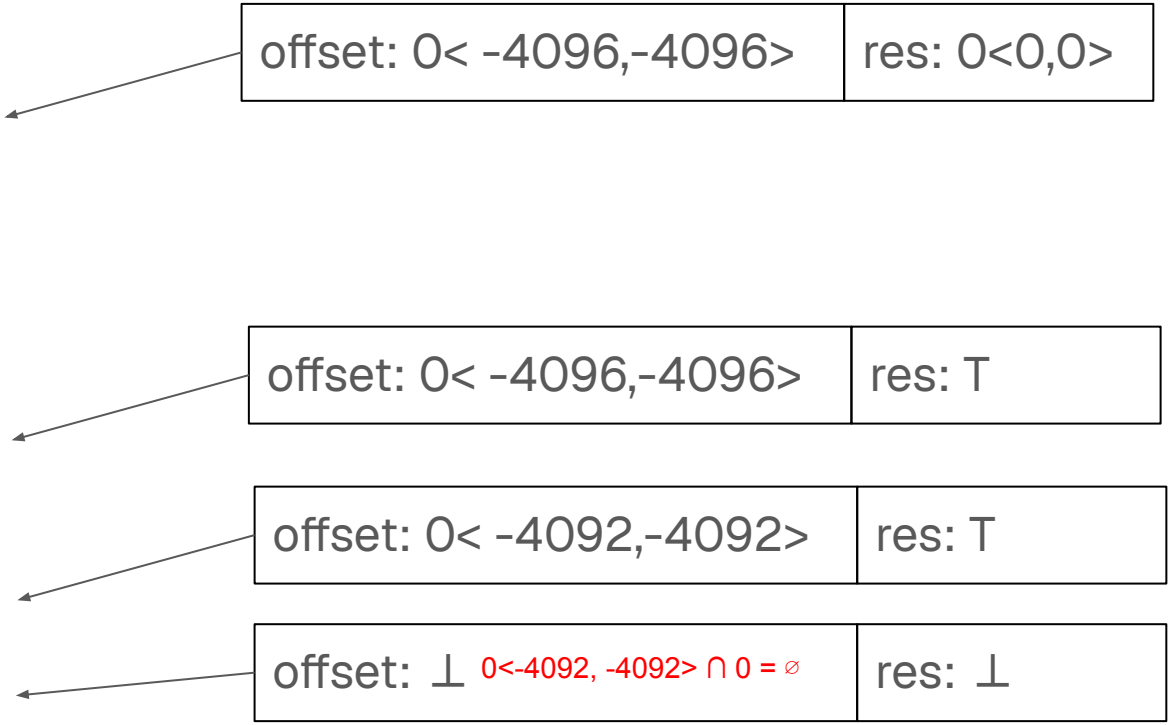
Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9       loop $loop_start
10        local.get $offset
11        i32.const 8192
12      i32.add
13      i32.load
14        local.get $offset
15        i32.const 4096
16      i32.add
17      i32.load
18    i32.mul
19      local.get $res
20    i32.add
21    local.set 0
22      local.get $offset
23      i32.const 4
24    i32.add
25    local.tee $offset
26    br_if $loop_start
27    end
28  local.get $res
29  )
```



Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9         loop $loop_start
10              local.get $offset
11              i32.const 8192
12            i32.add
13            i32.load
14              local.get $offset
15              i32.const 4096
16            i32.add
17            i32.load
18          i32.mul
19            local.get $res
20        i32.add
21        local.set 0
22              local.get $offset
23              i32.const 4
24            i32.add
25          local.tee $offset
26        br_if $loop_start
27        end
28    local.get $res
29  )
```



Let's walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9         loop $loop_start
10              local.get $offset
11              i32.const 8192
12            i32.add
13            i32.load
14              local.get $offset
15              i32.const 4096
16            i32.add
17            i32.load
18          i32.mul
19            local.get $res
20        i32.add
21        local.set 0
22              local.get $offset
23              i32.const 4
24            i32.add
25          local.tee $offset
26        br_if $loop_start
27        end
28    local.get $res
29  )
```

0<-4092, -4092>.remove(0) = 0<-4092, -4092>

0<-4096, -4096> ∪ 0<-4092, -4092> 0<0, 0> ∪ T = T

offset: 4< -4096,-4092>	res: T

offset: 0< -4096,-4096>	res: T

offset: 0< -4092,-4092>	res: T

offset: ⊥	res: ⊥

Let’s walk through the dot product program:

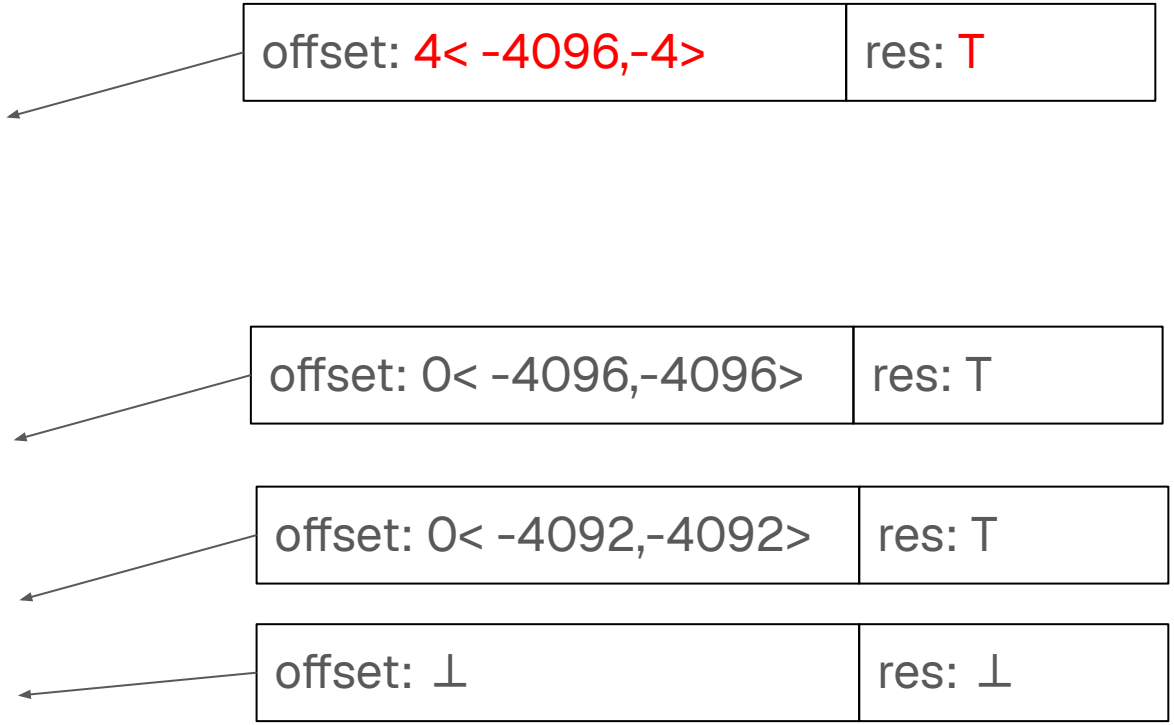
```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9       loop $loop_start
10        local.get $offset
11        i32.const 8192
12      i32.add
13      i32.load
14        local.get $offset
15        i32.const 4096
16      i32.add
17      i32.load
18    i32.mul
19      local.get $res
20    i32.add
21    local.set 0
22      local.get $offset
23      i32.const 4
24    i32.add
25    local.tee $offset
26    br_if $loop_start
27    end
28  local.get $res
29  )
```

Thresholds: [-4096, -1, 0, 4, 4096, 8192]

offset: 4< -4096,-1>	res: T
offset: 0< -4096,-4096>	res: T
offset: 0< -4092,-4092>	res: T
offset: ⊥	res: ⊥

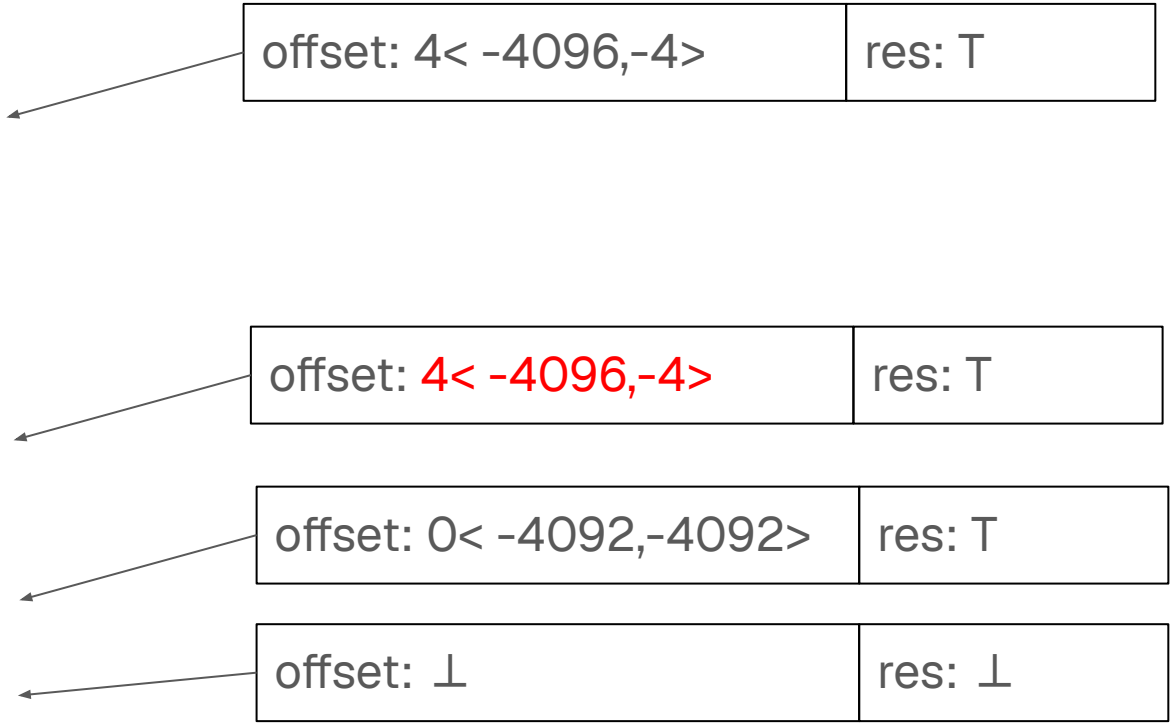
Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9       loop $loop_start
10        local.get $offset
11        i32.const 8192
12      i32.add
13      i32.load
14        local.get $offset
15        i32.const 4096
16      i32.add
17      i32.load
18    i32.mul
19      local.get $res
20    i32.add
21    local.set 0
22      local.get $offset
23      i32.const 4
24    i32.add
25    local.tee $offset
26    br_if $loop_start
27    end
28  local.get $res
29  )
```



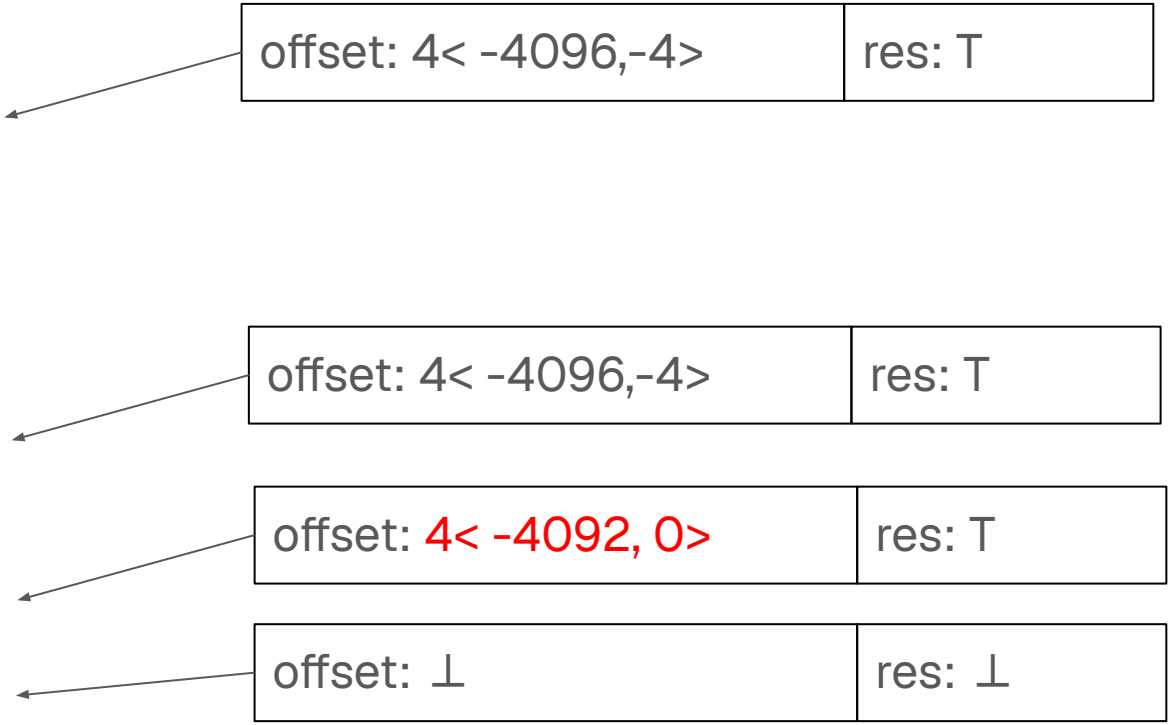
Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9       loop $loop_start
10        local.get $offset
11        i32.const 8192
12      i32.add
13      i32.load
14        local.get $offset
15        i32.const 4096
16      i32.add
17      i32.load
18    i32.mul
19      local.get $res
20    i32.add
21    local.set 0
22      local.get $offset
23      i32.const 4
24    i32.add
25    local.tee $offset
26    br_if $loop_start
27    end
28  local.get $res
29  )
```



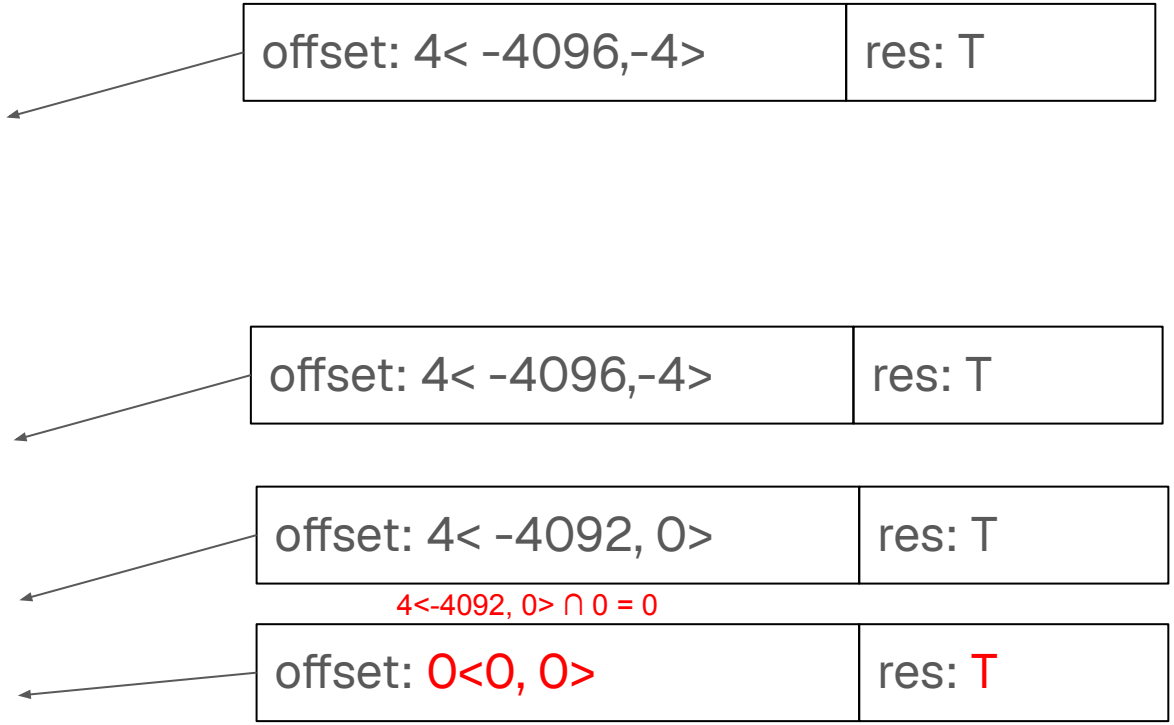
Let's walk through the dot product program:

```
1        (memory 1)
2     (func $dotproduct (result i32)
3       (local $res i32)
4       (local $offset i32)
5         i32.const 0
6       local.set $res
7         i32.const -4096
8       local.set $offset
9           loop $loop_start
10                local.get $offset
11                i32.const 8192
12              i32.add
13              i32.load
14                local.get $offset
15                i32.const 4096
16              i32.add
17              i32.load
18            i32.mul
19              local.get $res
20          i32.add
21          local.set 0
22                local.get $offset
23                i32.const 4
24            i32.add
25            local.tee $offset
26          br_if $loop_start
27          end
28      local.get $res
29    )
```



Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9         loop $loop_start
10              local.get $offset
11              i32.const 8192
12            i32.add
13            i32.load
14              local.get $offset
15              i32.const 4096
16            i32.add
17            i32.load
18          i32.mul
19            local.get $res
20        i32.add
21        local.set 0
22              local.get $offset
23              i32.const 4
24            i32.add
25          local.tee $offset
26        br_if $loop_start
27        end
28    local.get $res
29  )
```



Let’s walk through the dot product program:

```
1   (memory 1)
2   (func $dotproduct (result i32)
3     (local $res i32)
4     (local $offset i32)
5       i32.const 0
6     local.set $res
7       i32.const -4096
8     local.set $offset
9       loop $loop_start
10        local.get $offset
11        i32.const 8192
12      i32.add
13      i32.load
14        local.get $offset
15        i32.const 4096
16      i32.add
17      i32.load
18    i32.mul
19      local.get $res
20    i32.add
21    local.set 0
22      local.get $offset
23      i32.const 4
24    i32.add
25    local.tee $offset
26    br_if $loop_start
27    end
28  local.get $res
29  )
```

Fix point!

4<-4092, 0>.remove(0) = 4<-4092, -4>

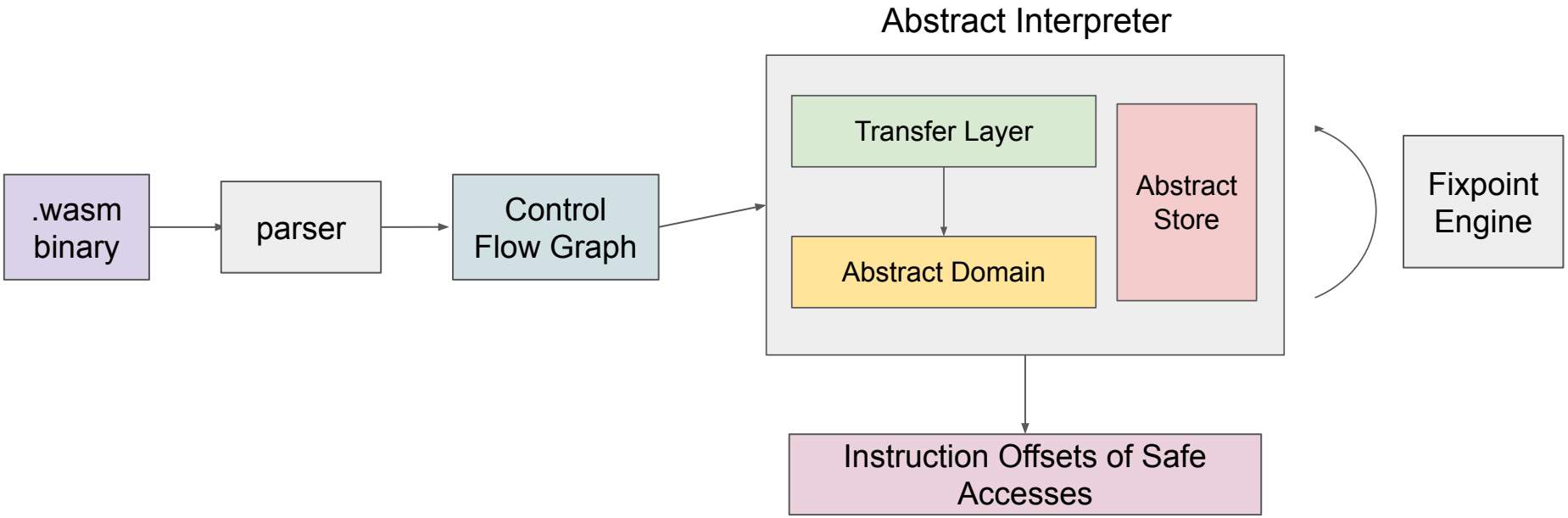
offset: 4< -4096,-4>	res: T

offset: 4< -4096,-4>	res: T

offset: 4< -4092, 0>	res: T

offset: 0<0, 0>	res: T

Architecture of WasmBounds



How can we be more confident that WasmBounds does not make false positives?

Two strategies so far:

1) **Verify the soundness of the abstract domain with Verus**

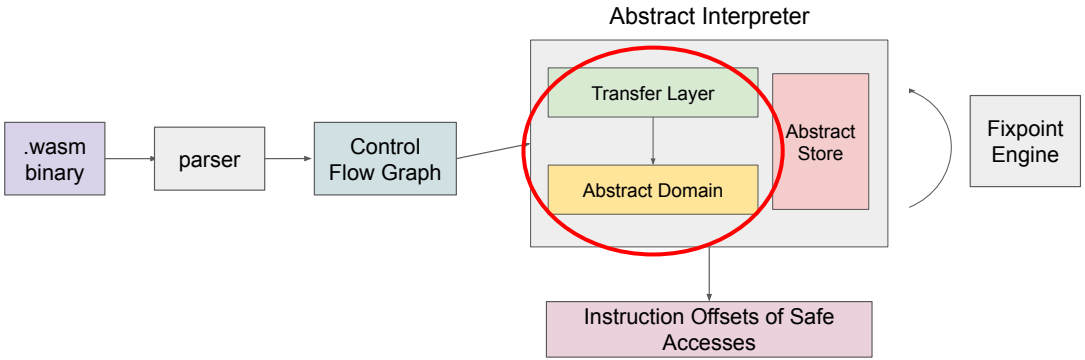
2) **Make the transfer layer infer Wasm semantics from SpecTec**

Two strategies so far:

1) Verify the soundness of the abstract domain with Verus

2) Make the transfer layer infer Wasm semantics from SpecTec

- Similar to Mikail Khan and Ben Titzer's Canonical Bytecode Definitions



What does “soundness” of the abstract domain mean?

Operations on the abstract intervals over-approximates the corresponding operation on the associated concrete set.

For example,

- $\gamma(A) \oplus \gamma(B) \subseteq \gamma(\text{add}(A, B))$
- $\gamma(A) \cap \gamma(B) \subseteq \gamma(\text{meet}(A, B))$
- $\gamma(A) \cup \gamma(B) \subseteq \gamma(\text{join}(A, B))$
- $\text{contains}(A, B) \Rightarrow \gamma(B) \subseteq \gamma(A)$

How does Verus verify that the domain is sound?

```
pub fn join(&self, joinee: &Self) -> (result: Self)
    requires
        self.wf(),
        joinee.wf(),
    ensures
        result.wf(),
        forall|value: u32|
            self.member(value) || joinee.member(value) ==> result.member(value),
```

Extract ground truth out of SpecTec

According to the spec,

```
i32.add:
    c1 := stack.pop()
    c2 := stack.pop()
    stack.push(iadd(c1, c2))
```

Replace iadd() with AbstractI32::add()

Conclusion

Abstract interpretation is a viable method for recovering safe memory accesses from a Wasm binary.

Thank you for listening!

Contact me at:
emmasudo@cs.stanford.edu
emmasudo.com