EPFL

Symbolic LLVM Memory Sandboxing for Safe and Deterministic WebAssembly-Based Execution

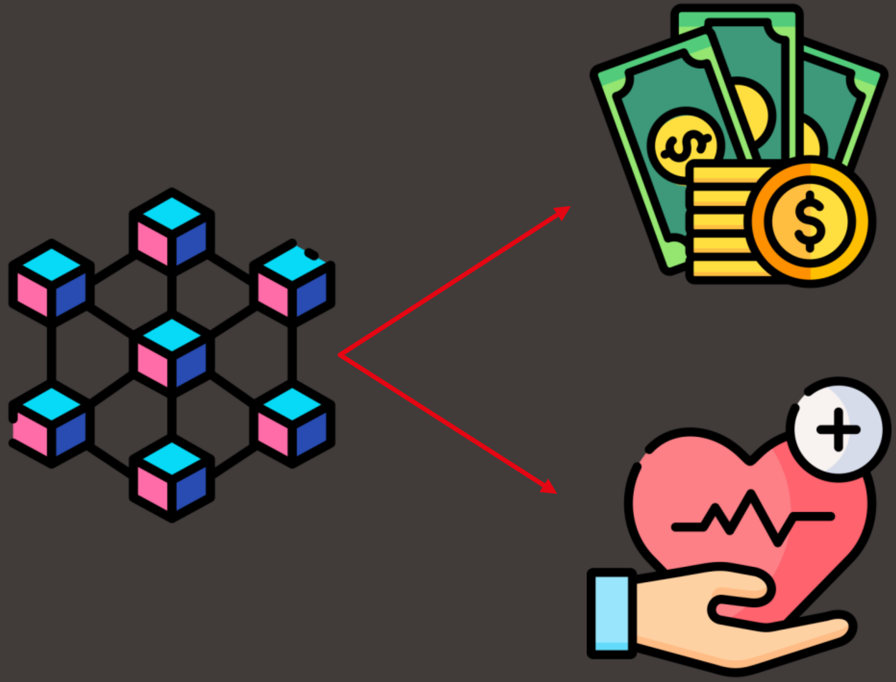
Xavier Ogay – Spring 2025

Under the supervision of Gauthier Voron

Xavier Ogay

Smart Contracts: Code That Controls Real Money

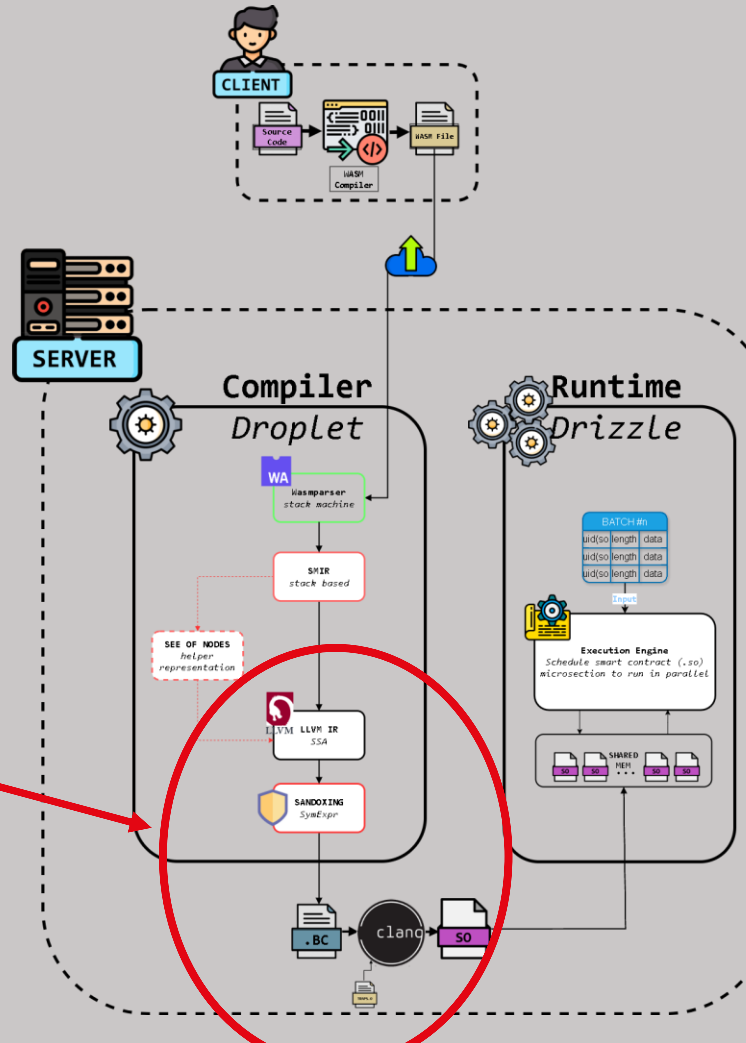
- **Blockchain** powers *critical sectors*:
  - finance,
  - healthcare,
  - identity, …
- State Machine Replication ensures **same state**.
- **Determinism** is non-negotiable: divergence = lost funds or broken logic.



EPFL

Project Context — Safe and Fast Smart Contract Execution

- **Droplet:** compiler for *WASM* smart contracts
- **Drizzle:** Runtime for parallel, deterministic execution
- Goal: **sandbox memory** with **min sacrifice of performance**



EPFL

Contribution — Symbolic Memory Sandboxing

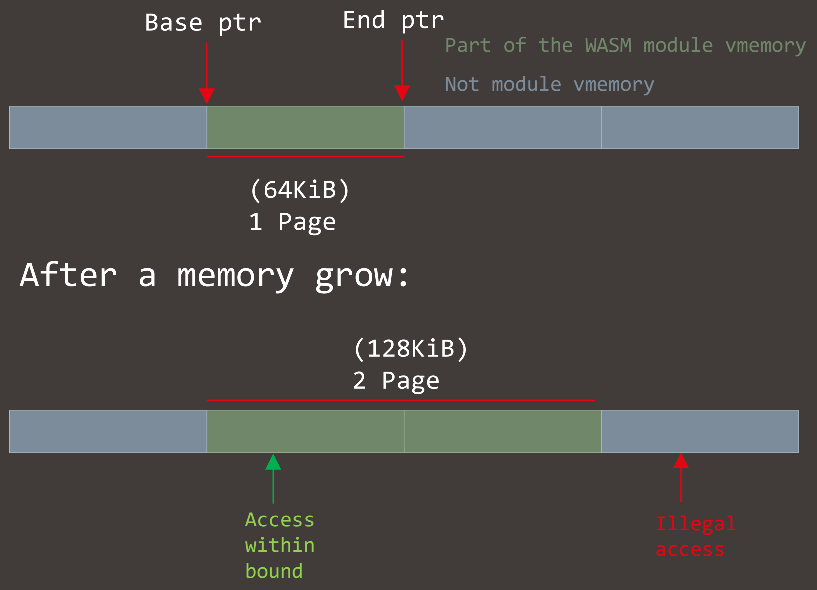
- **SymExpr**: Canonical memory reasoning via *symbolic expressions*
- **SymbolicState**: Track & merge state across blocks
- **Optimized Checks**: Hoist, deduplicate, and group bounds checks
- Up to **85%** overhead reduction on benchmarks

Xavier Ogay

WebAssembly Memory Model — Page-Based Linear Memory

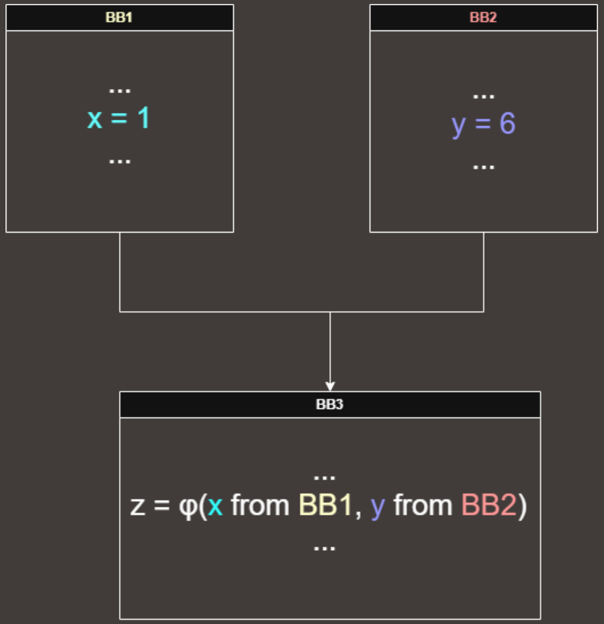
- Flat, linear memory: a contiguous **array of i8 bytes**
- Grows in units of **64 KiB pages** (via *memory.grow*)
- Explicit bounds checks needed, out-of-bounds = trap

Memory accesses needs to be in range **[Base ptr ; End ptr]**



Phi Nodes — Merging Values at Control Flow Joins

- Used in **SSA** form to merge values at CF joins
- ϕ(*v_x* from *BBx*, *v_y* from *BBy*) selects based on incoming path
- Compile time: must assume both values are possible



z may be x or y – **check(x,y)**

Memory Safety Strategies — From Naive to Loop-Aware

- **Naive**: Check every memory access — easy but **slow**
- **Opt1**: Skip checks for addresses already validated
- **Opt2**: Group & hoist checks using loop-aware analysis
- **Opt3**: Shared Check at Block Entry

```
BB1

ptr1 = base + 8
check(ptr1)
load ptr1

ptr2 = base + 8
check(ptr2)
load ptr2
```

```
BB1

ptr1 = base + 8
check(ptr1)
load ptr1

ptr2 = base + 8
-- check skipped (already validated)
load ptr2
```

```
BB Loop Header

i = 0
check(base, base + N * 8)
-- hoisted check for whole access range
```

```
BB Loop Body

ptr = base + i * 8
load ptr
i = i + 1
br if i < N to BB_loop_body
```

```
BB grouped

check(base, base + 24)

ptr1 = base + 0
load ptr1

ptr2 = base + 8
load ptr2

ptr3 = base + 16
load ptr3
```

SymExpr — Symbolic Memory Algebra

- Instructions as **symbolic algebra**
- **Canonical & normalized** → enable equivalence, deduplication

```
> Instruction Expressions:
    " %6 = lshr i64 %1, 3" => (v0x5db8c7409538 >> 3)
    " %10 = load ptr, ptr @1, align 8" => [v0x5db8c73fdbc0]%0
    " %13 = add i64 %12, 1" => (1 + [([v0x5db8c73fdbc0]%0 + v0x5db8c7407fc8)]%0)
    " %12 = load i64, ptr %11, align 4" => [([v0x5db8c73fdbc0]%0 + v0x5db8c7407fc8)]%0
    " %14 = add i64 %8, 8" => (8 + v0x5db8c7407fc8)
    " %15 = add i64 %9, -1" => (v0x5db8c7408068 - 1)
    " %11 = getelementptr i8, ptr %10, i64 %8" => ([v0x5db8c73fdbc0]%0 + v0x5db8c7407fc8)
    " store i64 %13, ptr %11, align 4" => ([v0x5db8c73fdbc0]%0 + v0x5db8c7407fc8)
    " %9 = phi i64 [ %6, %5 ], [ %15, %7 ]" => v0x5db8c7408068
    " %8 = phi i64 [ %0, %5 ], [ %14, %7 ]" => v0x5db8c7407fc8
```

Canonicalization

$$Expr_1 = a + b, \quad Expr_2 = b + a$$

$$Canon(Expr_1) = Canon(Expr_2) = a + b$$

Normalization

$$Expr = 3 \cdot i + 4 \cdot j + 2 + i + 8$$

$$Norm(Expr) = 4 \cdot i + 4 \cdot j + 10$$

SymbolicState — Tracking Symbolic Semantics

Xavier Ogay

- Tracks symbolic memory and **value info** per **basic block**
- Fields:
  -value_exprs,
  -memory_accesses,
  -assumptions, etc…
- Propagates across control flow with merging
- Enables loop-aware memory check optimization

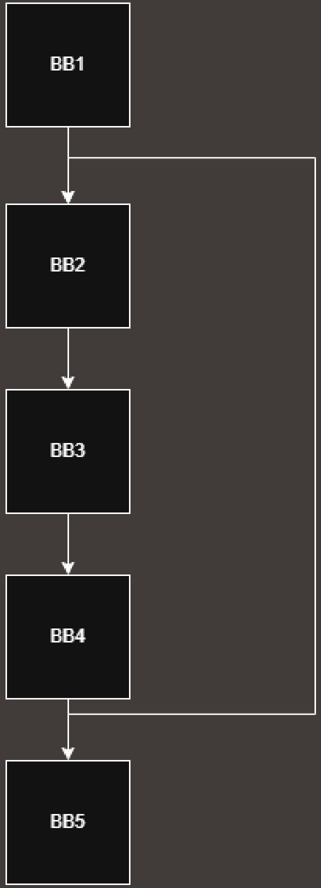
```
Block: bb_3
=== Symbolic State ===

> Value Expressions:
    "  %8 = phi i64 [ %0, %5 ], [ %14, %7 ]" => v0x5b77bd675fc8
    "i64 -1" => -1
    "  %9 = phi i64 [ %6, %5 ], [ %15, %7 ]" => v0x5b77bd676068
> Instruction Expressions:
    "  %9 = phi i64 [ %6, %5 ], [ %15, %7 ]" => v0x5b77bd676068
    "  %14 = add i64 %8, 8" => (8 + v0x5b77bd675fc8)
    "  %11 = getelementptr i8, ptr %10, i64 %8" => ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
> Memory Accesses State:
    [[v0x5b77bd66bbc0]%0] => Range: [v0x5b77bd66bbc0]%0 ..= [v0x5b77bd66bbc0]%0

> Checked Address Ranges:

> Assumptions:

> Induction Variables:

> Memory addr accessed:
    "  %12 = load i64, ptr %11, align 4" => ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
    "  store i64 %13, ptr %11, align 4" => ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
    "  %10 = load ptr, ptr @1, align 8" => v0x5b77bd66bbc0
- AccessPatternGroup:
  Base: ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
  Offsets:
    - 0

- AccessPatternGroup:
  Base: v0x5b77bd66bbc0
  Offsets:
    - 0


Store Counter: 1
```

Loop-Aware Memory Check Optimization

- Loop detected via **dominator + back edges**
- Loop merge → **fixed-point**
- Phi-resolved for accessed memory
- Pre-loop **check**: BB guard total memory range

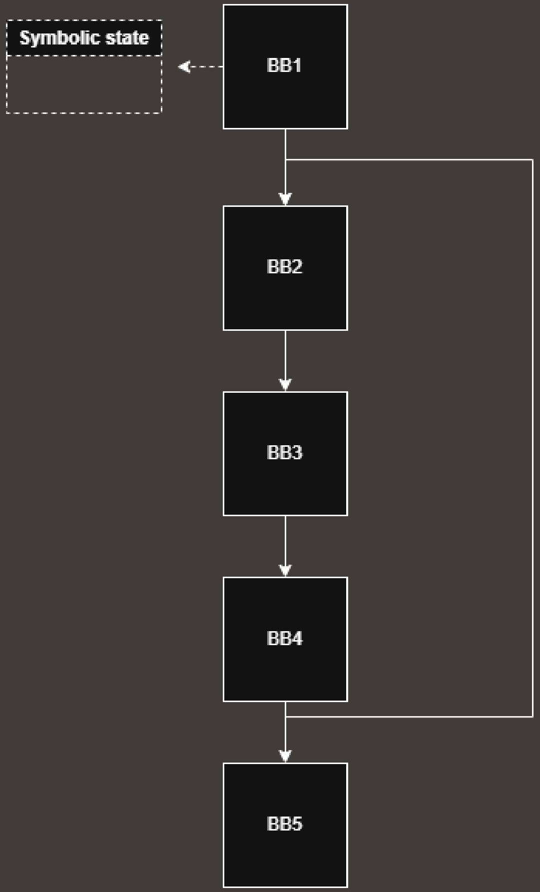
```
LoopMemoryContext:
==== Loop Memory Context ====
Header:
Loop Induction Var: v0x5b77bd676068 ∈ [(v0x5b77bd677538 >> 3) .. (v0x5b77bd676068 - 1)] by -1
All Induction Variables:
    v0x5b77bd675fc8 ∈ [v0x5b77bd677510 .. Addition from (8 + v0x5b77bd675fc8)]
    v0x5b77bd676068 ∈ [(v0x5b77bd677538 >> 3) .. Substraction from (v0x5b77bd676068 - 1)]
Step Expression: Substraction from (v0x5b77bd676068 - 1)
Bound: 0

> Memory Expressions:
    "  %12 = load i64, ptr %11, align 4"
    "  %10 = load ptr, ptr @1, align 8"
    "  store i64 %13, ptr %11, align 4"

> Induction Related Memory Expressions:
    ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
AccessPatternGroup:
  Base: ([v0x5b77bd66bbc0]%0 + v0x5b77bd675fc8)
  Offsets:
    - 0

(v0x5b77bd677538 >> 3)
Estimated symbolic range = (v0x5b77bd677538 >> 3) with (v0x5b77bd677538 >> 3) steps (step -1)
```

Loop-Aware Memory Check Optimization



For each function:
- Build the **CFG**
- Build **the Post Dominator Tree**

Loop-Aware Memory Check Optimization

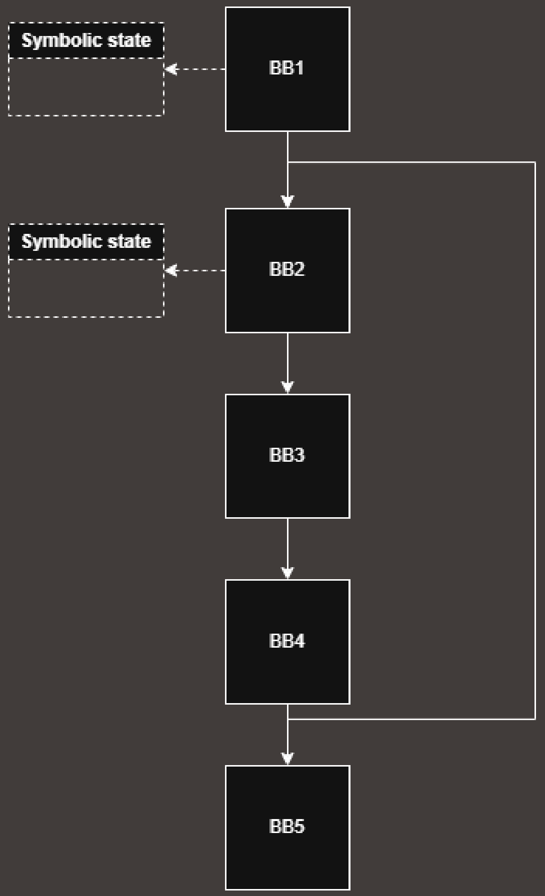


For each basic block in function:

- Build the **Symbolic State**
- Traverses CFG in **reverse post-order** to merge prior block info

Loop-Aware Memory Check Optimization

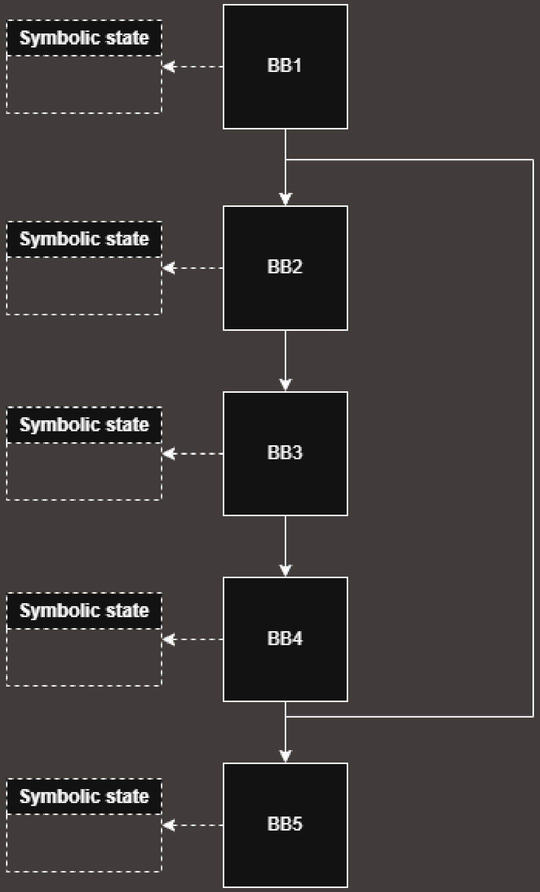
Xavier Ogay



For each basic block in function:

- Build the **Symbolic State**
- Traverses CFG in **reverse post-order** to merge prior block info

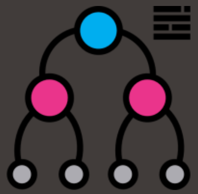
Loop-Aware Memory Check Optimization



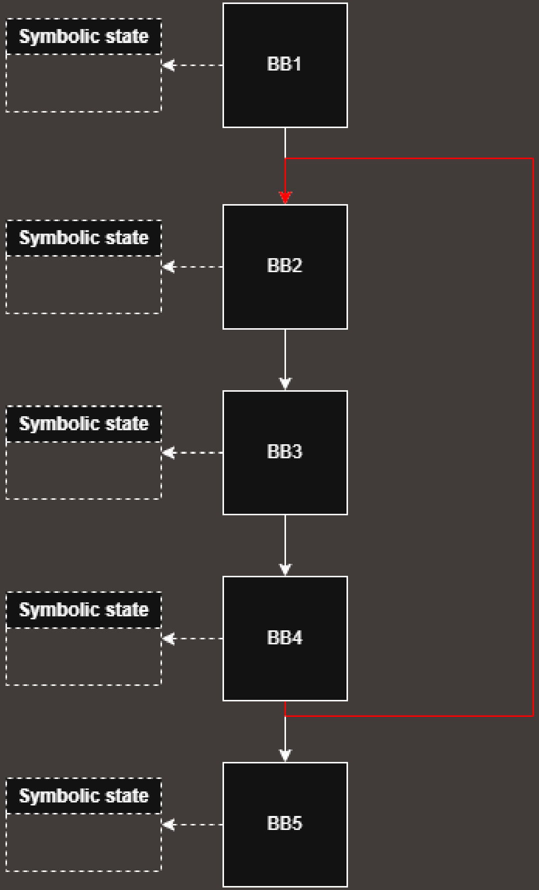
For each basic block in function:

- Build the **Symbolic State**
- Traverses CFG in **reverse post-order** to merge prior block info

Loop-Aware Memory Check Optimization



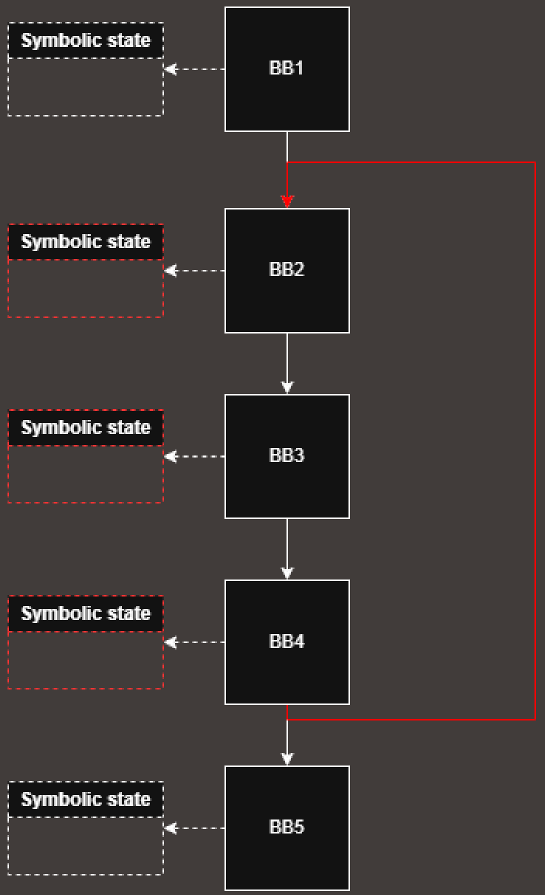
Use the **Dominator tree**



Loop detection:

- **Back edges** in CFG: edges where target **dominates** the source
- Header = target, tail = source → defines loop boundary
- Reachable blocks → forms the **natural loop body**

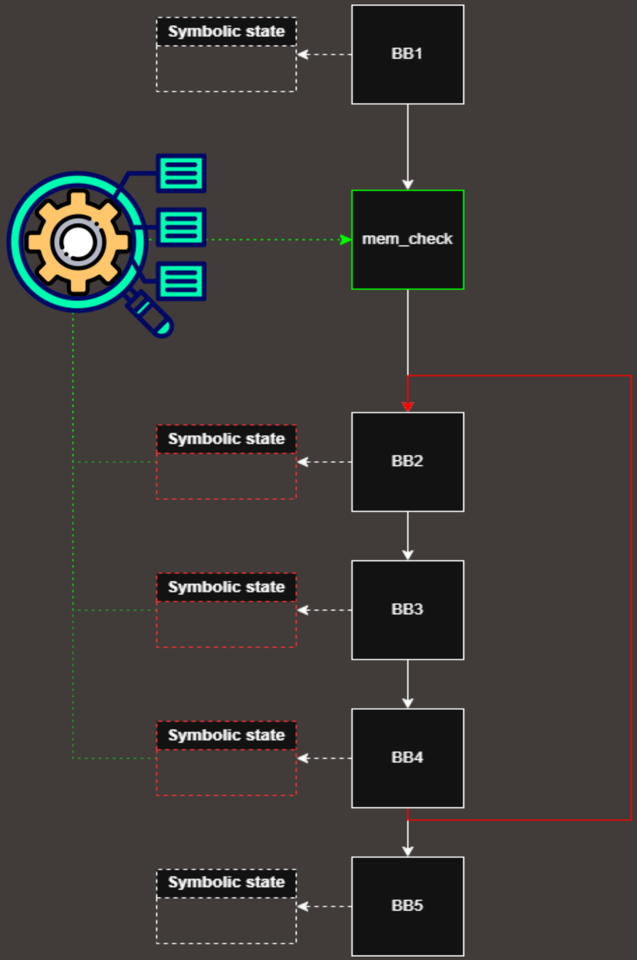
Loop-Aware Memory Check Optimization



Loop refinement:

- **Fixed-point** until state stabilizes or hits limit
- Detect induction vars and step (e.g. *i += 1*)

Loop-Aware Memory Check Optimization



Extract induction related **access pattern**

Check block emission:

- Detect loop-strided memory **access patterns**
- **Group accesses** by symbolic base/stride
- Insert pre-loop block with **range checks** per group

Xavier Ogay

Assumption-Based Memory Check Elision (Skeleton)

User code

```
i = 0
...
if (i < 4) {
    access(i + 8)
}
if (j >= 6) {
    access(j + 8)
}
```

Assumptions

i ∈ [0, 3]

j ∈ [6, ∞]

Access Range *i*: i + 8 ∈ [8, 11]
Access Range *j*: j + 8 ∈ [14, ∞)
↳ Proven *disjoint*
→ no *overlap* → **two check needed***

Other possibility
→ *overlap* → **maybe can be merged**
→ subset of other → **emit one check larger range**

- Track constraints from user instructions (e.g. *icmp*)
- Map SymExpr to **min/max value assumptions**
- Not evaluated, as not fully functional

Symbolic Instrumentation in Droplet Improves Execution Performance

- `Benchmarked 12 kernels under 4 configurations`
- `Naive checks:` **`1.5× to 10×`** `slowdown`
- `Optimized: up to` **`85% overhead reduction`**
- `Tested under realistic` *`.so`* `batching`

Benchmark	No sandbox [µs]	Check (naive) [µs]	Opt1 [µs]	Opt2 [µs]	Opt3 [µs]	SU (check→opt1)	SU (check→opt2)	SU (check→opt3)
2d	0.47 ± 0.35	1.36 ± 0.45	0.94 ± 0.39	0.54 ± 0.24	0.52 ± 0.21	31%	60%	62%
add1	0.35 ± 0.37	1.47 ± 0.85	0.76 ± 0.51	0.37 ± 0.36	0.37 ± 0.29	48%	75%	75%
addbounded	2.75 ± 0.43	29.04 ± 6.42	16.42 ± 1.92	4.42 ± 1.53	4.51 ± 1.64	43%	85%	84%
conditional	1.75 ± 0.66	2.78 ± 2.28	2.74 ± 0.96	2.26 ± 0.86	–	2%	19%	–
fibonaccilike	0.44 ± 0.24	1.29 ± 0.47	1.30 ± 0.69	0.58 ± 0.64	–	−1%	55%	–
matrix	2.58 ± 3.41	7.41 ± 3.35	7.39 ± 3.33	2.60 ± 4.32	–	0%	65%	–
nested	0.37 ± 0.49	1.75 ± 0.61	0.83 ± 0.62	0.49 ± 0.61	0.48 ± 0.47	53%	72%	73%
prefix	0.49 ± 0.44	1.56 ± 0.74	0.94 ± 0.57	0.48 ± 0.33	–	40%	69%	–
redundant	0.36 ± 0.34	1.47 ± 0.74	0.76 ± 0.38	0.38 ± 0.42	0.36 ± 0.16	49%	74%	75%
reverse	0.39 ± 0.27	1.18 ± 0.56	0.78 ± 0.62	0.40 ± 0.52	0.38 ± 0.26	34%	66%	68%
slidewindow	0.66 ± 0.49	1.62 ± 0.67	1.60 ± 0.57	–	0.71 ± 0.27	1%	–	56%
stride	0.29 ± 0.13	0.67 ± 0.30	0.47 ± 0.31	0.33 ± 0.43	0.32 ± 0.29	30%	51%	52%

Conclusion — Efficient and Safe Memory Sandboxing

- WebAssembly smart contracts -> **deterministic**, want **safe execution**
- Runtime sandboxing is too costly for performance-critical workloads
- Our solution: **static symbolic memory analysis** in **Droplet**
- **Reduces redundant checks** while maintaining spatial safety
- Achieves significant **overhead elimination** on complex loop-heavy code