DCL - Distributed Computing Lab
Prof. Guerraoui Rachid
Main Supervisor
Dr. Voron Gauthier
Project Supervisor

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

Xavier Ogay - 301681 - *xavier.ogay@epfl.ch*

Abstract

This work presents a symbolic sandboxing framework for securely and efficiently executing WebAssembly-based smart contracts within a deterministic, replicated execution model. The system targets Droplet, an ahead-of-time compiler producing shared objects from WASM modules, and enhances it by statically emitting memory safety checks at the LLVM IR level. By leveraging symbolic expressions to reason about memory access patterns, especially in loops and across multiple basic blocks, the framework reduces redundant runtime instrumentation while preserving strong spatial safety guarantees.

Keywords: WebAssembly, symbolic expression, LLVM, memory sandboxing, smart contracts, static analysis

Introduction

Decentralized systems such as blockchains rely on replicated state machine execution to ensure consistency across mutually untrusted nodes. Every node re-executes submitted transactions deterministically, enforcing convergence on a global state. This execution model secures real-world assets—including money, property, and identities—but it imposes strict correctness guarantees: execution must be deterministic. Any deviation in execution results across replicas may lead to failure—potentially resulting in permanent financial loss or corrupted state within the blockchain.

Historically, consensus protocols were the dominant performance bottleneck. However, advances in high-throughput consensus algorithms and transaction parallelism have shifted the limiting factor to the execution layer. In this new landscape, the cost of safely executing smart contracts has become a primary concern.

Figure 1 outlines the end-to-end pipeline targeted for smart contract compilation and execution. Clients author contracts in their preferred source language and compile them to WebAssembly (WASM), which acts as a portable, sandboxed intermediate representation. WASM guarantees determinism across diverse language frontends and serves as the canonical entry point for further processing.

On the server side, the **Droplet** ahead-of-time (AOT) compiler takes over. It parses WASM into an internal stack-based intermediate representation called SMIR, which introduces basic block structure while retaining stack discipline. SMIR is then translated into LLVM IR, a static single assignment (SSA) form. The resulting LLVM bitcode is compiled to a `.so` shared object, which encapsulates the native code version of the smart contract.

The final `.so` file is handed off to **Drizzle**, a WIP runtime system responsible for scheduling contract execution. Drizzle operates in batch mode, using microsection analysis to execute multiple smart contract calls in parallel when their memory regions do not conflict.

We focuse on extending **Droplet** with a symbolic sandboxing framework for static memory safety enforcement. All contributions are integrated directly into the Droplet compiler pipeline, operating at the LLVM IR level. The objective is to emit memory bounds checks statically during compilation—rather than dynamically at runtime—thereby reducing overhead while upholding the determinism and spatial safety guarantees essential in replicated execution environments.

The initial goal of this project was to explore **Sea of Nodes (SoN)** representations as a foundation for precise memory check placement. However, given the technical complexity of rewriting Droplet's intermediate representations, combined with my limited prior experience in compilation, LLVM, and Rust, this proved too ambitious for the project timeline. As a result, the focus shifted to designing a symbolic analysis tool capable of tracking memory access patterns across functions and control-flow constructs, especially for loop-heavy and multi-block regions. The resulting symbolic infrastructure lays the groundwork for advanced memory reasoning and can later serve as a companion layer to a full SoN-based optimization system—forming a versatile toolset for safe and efficient memory instrumentation.

Overview and Contributions

We present a new memory sandboxing framework built into the Droplet compilation pipeline, targeting WebAssembly smart contracts compiled to LLVM IR. Our approach is based on **symbolic reasoning at compile time** over memory access patterns, enabling us to statically identify bounds and insert minimal, provably safe memory checks.

By representing memory addresses using symbolic expressions, tracking loop induction variables, and propagating constraints across control-flow, our system hoists and deduplicates bounds checks. This leads to substantial performance gains while preserving strict spatial safety.

Our main contributions are:

- **SymExpr:** A custom symbolic expression framework supporting canonicalized memory reasoning and analysis.
- **SymbolicState:** A basic block state with inter-block merging and propagation mechanisms.
- **Memory Check Optimization:** Loop-aware check hoisting and intra-block grouping to reduce instrumentation overhead.
- **Evaluation on Real Code:** Compilation and execution of test code, demonstrating up to 85% overhead elimination.

Background

Deterministic Replicated Execution

In replicated smart contract systems, such as blockchains, all nodes must execute the same code with identical results to maintain consensus. These systems follow the model of State Machine Replication, where each node

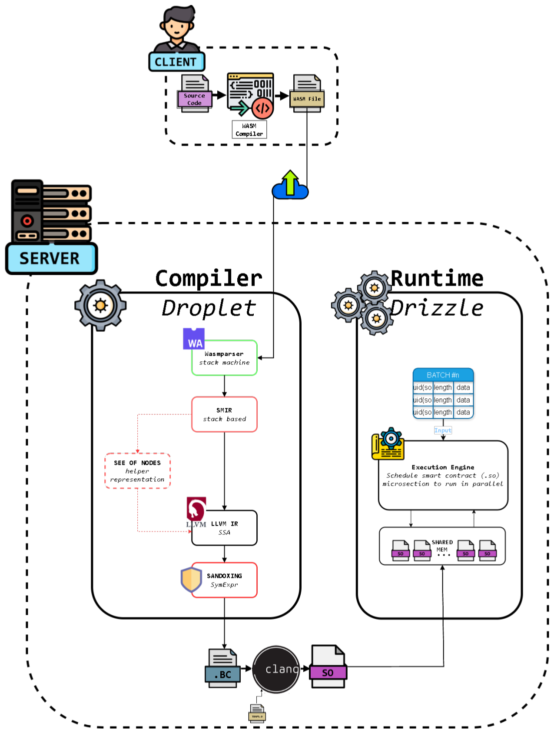


Figure 1: Droplet Context Overview

runs an identical deterministic program on a shared sequence of inputs. This ensures consistency across replicas even in adversarial or distributed environments. Non-determinism—especially from memory access violations or race conditions—can break this model, leading to diverging states and catastrophic failure.

WebAssembly and Its Linear Memory Model

WebAssembly (WASM) adopts a flat linear memory model: a single contiguous, byte-addressable region representing the module's memory. All load and store operations are expressed as offsets from this unified base, simplifying static analysis.

When lowered to LLVM IR, Wasm memory appears as `i8` arrays, enabling fine-grained reasoning about byte-level access patterns. This uniform layout, free from traditional allocator fragmentation, allows symbolic analysis and bounds check insertion to focus solely on base-relative offsets.

LLVM IR and SSA Form

LLVM IR represents programs in Static Single Assignment (SSA) form, where each variable is defined exactly once and every use refers to a unique definition. This property simplifies dataflow analysis, making it easier to track value provenance and transformations.

Symbolic Expression Concepts

Symbolic expressions (`SymExpr`) represent program values and memory addresses not as concrete numbers, but as algebraic expressions over variables. In our context, these expressions are constructed during compilation to model the result of computations such as arithmetic operations, pointer offsets, or loop-based indexing. Each expression captures not just a value, but the computation that produced it.

This symbolic representation enables the compiler to reason about the equality or equivalence of different expressions at compile time. For instance, if two memory accesses share the same symbolic address, they are guaranteed to refer to the same location, and a previously validated bounds check may be safely reused. Likewise, symbolic expressions can be compared to determine containment within memory regions, or to infer the full range of addresses accessed by a loop.

Phi Nodes and Symbolic Ambiguity

LLVM IR uses `phi` nodes to merge values from multiple control-flow paths into a single SSA variable. While essential for expressing loops and branching, phi nodes introduce ambiguity in symbolic analysis—each incoming value may correspond to a different symbolic expression. To maintain soundness, our system conservatively wraps such merged expressions using the `OneOf` construct, representing the union of all possible values. This conservative modeling ensures safety but may limit optimization opportunities if value disambiguation is not possible.

Control-Flow Graphs and Dominator Trees

A Control-Flow Graph (CFG) models the execution flow between basic blocks in a function. Dominator trees are derived from CFGs to identify blocks that must precede others on all paths. These structures are crucial in our system for loop detection, safe state propagation, and sound placement of hoisted memory checks.

Loop Detection and Induction Variables

Loops are identified as natural cycles in the CFG via back-edge detection. Within loops, induction variables are used to describe predictable iteration patterns. We leverage them to model symbolic memory access ranges and to emit single memory checks at loop headers, optimizing performance while maintaining safety.

Design

To introduces a symbolic sandboxing mechanism for securely executing WebAssembly-based smart contracts compiled into native `.so` modules, the core idea is to statically emit memory bounds checks during compilation to enforce spatial safety at runtime, ensuring all memory accesses stay within the expected sandboxed region. Thanks to WASM's linear memory model, emitting bounds checks is simpler, as all accesses are relative to a single contiguous base, unlike in buddy allocators where memory is partitioned and harder to track statically.

Memory Group Checking via LLVM IR

We target the LLVM IR emitted from WASM modules, leveraging its SSA form and structured control flow to reason about memory accesses. The goal is to determine the minimal and maximal bounds of access for each group of related memory operations and insert a single check guarding the entire group. This avoids redundant checks and improves performance while retaining safety guarantees.

Assumption-Based Check Elision (Preliminary)

An early design component of the symbolic tracking system included the notion of *assumptions*, which capture semantic conditions implied by the original LLVM IR code. These assumptions—such as bounds comparisons or pointer range tests—could be harvested from existing instructions like `icmp` and interpreted as constraints over `SymExpr` expressions. The vision was to allow such assumptions to be explicitly tracked in the symbolic state and leveraged to suppress redundant memory checks when a valid constraint implied safety.

In practice, while the infrastructure for recording assumptions (`assumptions` field in `SymbolicState`) is partially implemented, full integration into the check emission logic was deferred to ease the merging of `SymbolicState`. Development focused instead on loop hoisting and access pattern merging, especially within `opt2` and `opt3` modes, where the performance benefits were more immediate. Incorporating assumption-based reasoning remains a promising direction for reducing unnecessary instrumentation in future extensions.

Initial Strategy: Symbolic Deduplication

The first strategy was to track every `load` and `store` instruction individually. Using symbolic expressions, each memory address was represented symbolically, and checks were inserted only if an equivalent or covering check had not been previously emitted. While this approach reduced overhead compared to naive instrumentation, it struggled with loops: checks were frequently reinserted inside loop bodies, incurring performance penalties due to repeated validations.

Refined Strategy: Function-Wide Symbolic Analysis

The second strategy introduced function-wide symbolic state tracking and loop-aware optimization. For every function reachable from the `droplet_entry` entry point, we analyze its basic blocks in control-flow order, assigning each a *SymbolicState* representing known expressions, assumptions, and memory access intents.

These states are initially assigned independently to each basic block and then merged conservatively in reverse post-order across the control-flow graph (CFG) to accumulate symbolic context. Natural loops are detected using CFG analysis and dominator tree construction. After loop structures are identified, a fixed-point refinement process is applied to the symbolic states of all blocks within each loop, stabilizing access patterns and state across iterations.

To ensure correct refinement in the presence of nested loops, the analysis detects loop nesting hierarchies and processes loops in an inside-out manner. Innermost loops are refined first, allowing their stabilized symbolic states to inform the refinement of outer loops. This ordering guarantees that dependencies between nested structures are resolved accurately and that induction relationships are fully established before they are reused in surrounding contexts.

Only once this refinement is complete, memory accesses that depend on loop induction variables are analyzed for symbolic bounds. Access patterns are then summarized using range analysis, and a single bounds check for maximum and minimum access address is inserted outside the loop in a new block. This hoisting prevents redundant validations within the loop body and substantially reduces runtime overhead. An example of nested loop hoisting is illustrated on figure 2 .

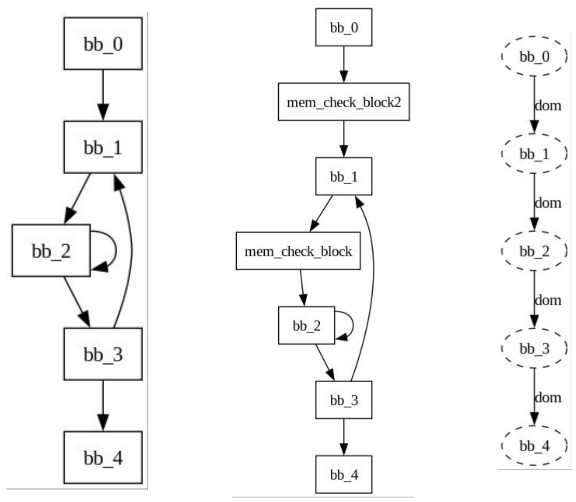


Figure 2: Control-flow graph of a nested loop function, its sandboxed variant, and the corresponding dominator tree.

Post-Loop and Residual Check Insertion

After loop optimization, all basic blocks are revisited. For blocks not involved in loops, memory accesses are grouped using symbolic pattern matching, and minimal/maximal range checks are inserted at the block entry. These checks guard all access in the group. For blocks that are part of loops but perform memory accesses not driven by loop induction, naive memory checks are inserted.

In future extensions, failing blocks could fall back to the original `mem_check` instrumentation, as seen in prior work [1], enabling precise error reporting. Currently, if symbolic reasoning fails to prove safety, compilation fails, ensuring correctness is never compromised.

Implementation

Memory Check Runtime Stub

Memory accesses are guarded using a dedicated runtime check function from previous work [1] emitted into the LLVM IR of each compiled module. As shown in Listing 1, the `mem_check` function receives three pointers: the memory base, the offset, and the target access address. It computes the effective access pointer using a GEP operation, then verifies that the target lies within the permitted bounds.

This function is emitted as a reusable subroutine rather than inlined by default. This design choice enables easier transformation and potential reuse across multiple access sites. In particular, it facilitates inter-pass check elision: redundant checks can be removed by identifying identical call sites or equivalent preconditions.

However, to balance performance, the system supports an `inline-memory-check` flag. When enabled, this annotates the `mem_check` function to be preferentially inlined during code generation. Inlining can eliminate function call overhead and improve branch prediction, especially in tight loops or frequently executed paths, while preserving the logical structure for optimization passes.

Listing 1: LLVM memory check function

```
define void @mem_check(ptr %0, ptr %1, ptr %2) {
  %4 = load ptr, ptr %0, align 8
  %5 = load i64, ptr %1, align 4
  %6 = getelementptr i8, ptr %4, i64 %5
  %lower = icmp uge ptr %2, %4
  %upper = icmp ult ptr %2, %6
  %cond = and i1 %lower, %upper
  br i1 %cond, label %common.ret, label %error
}
```

Symbolic Expression (SymExpr)

The core abstraction enabling static memory check reasoning in our toolchain is the symbolic expression. `SymExpr` defines a symbolic algebra over memory-relevant computations, capturing value computations symbolically rather than concretely.

The model includes:

- `Const(i64)`: Concrete literals.
- `Var(String)`: SSA variables or Phi variables.
- `Add`, `Sub`, `Mul`, `Div`, `ShiftL`, `Lshr`, `And`: Arithmetic and bitwise operations.

- `Min`, `Max`: Bounds expression modeling.
- `Load(SymExpr, u32)` and `Store(SymExpr)`: Symbolically abstract memory operations. Each `Store` represents a write to a symbolic address. The `Load` expression is annotated with a unique counter that captures the store epoch—the current value of the global `store_counter` at the time the load is analyzed. This counter increases conservatively after any memory write operation with potential overlapping address(e.g., a `Store`, a call to an unknown function, ensuring that loads are uniquely tied to the visible memory state at their point of occurrence. This mechanism enables intra-function memory disambiguation: when optimizing memory checks, two loads to the same symbolic address but under different store epochs are treated as potentially observing different memory states, and thus cannot be deduplicated. Those are build along the *SymbolicState* progress in basic blocks exploration explained in memory_accesses.
- `OneOf(Vec)`: Represents conditional symbolic expressions used for certain case of state merging

This representation allows us to statically model most pointer arithmetic and memory access ranges of the current *Droplet* outputs. Expressions are canonicalized: for example, `a + b` is ordered lexically to ensure commutativity, and `a * 1` simplifies to `a`. These transformations enable semantic deduplication during check insertion by ensuring that equivalent expressions are structurally identical.

To further enhance this capability, expressions are normalized into a form that approximates linear arithmetic when possible. For instance, symbolic forms like `3 * i + 4 * j + 10` are internally reduced into structured linear combinations, which aids in equivalence testing, offset analysis, and range evaluation.

Equality is implemented structurally through `PartialEq` and `Hash`, backed by string representations of canonical forms. This approach allows the use of set-like structures (e.g., `checked_sym_exprs`) to track previously validated memory accesses. However, maintaining this canonicalization becomes increasingly complex as new operations (e.g., `ShiftL`, `Min`, `OneOf`) are introduced. Each addition requires consistent handling in both expression simplification and string representation, creating a nontrivial engineering cost. Still, this rigor is essential for achieving safe and sound symbolic analysis throughout the compilation pipeline.

Assumptions and ValueRange

To improve the accuracy and applicability of symbolic memory checks, the system introduces the concept of *assumptions*—optional constraints associated with symbolic variables that define their possible value bounds. These are encoded as a `RangeAssumptions` map, associating each variable name with a tuple of minimum and maximum values (if known). For example, an assumption such as `x` $\in$ `[0, 10]` allows the symbolic analysis to reason concretely about expressions involving `x`.

The `ValueRange` abstraction encapsulates symbolic lower and upper bounds for a memory address or group access range. Using assumptions, the system can perform:

- **Containment checks:** determine whether a symbolic expression lies within a symbolic range.
- **Overlap checks:** test whether two symbolic ranges intersect or are disjoint.
- **Memory fit validation:** verify that a symbolic range fits within a preconfigured memory size.

These operations support both *approximate* and *exact* modes. Approximate reasoning provides conservative under- and over-estimates using partial bounds inferred from assumptions. Exact reasoning requires complete bounds for all participating variables and enables precise verification of symbolic constraints.

This framework enables more aggressive memory check elimination. When assumptions derived from the control-flow context (e.g., conditional branches, loop guards) imply that a memory access is safe, the corresponding `mem_check` call may be statically removed.

Currently, the system resolves only simple cases such as constant-bounded variables or expressions that simplify directly (e.g., 10 > x > 1). However, with further integration effort, it could be linked to an external symbolic inequality solver API to support richer inference and constraint propagation.

Note: This assumption-based simplification framework is not enabled in evaluation mode due to time constraints in development —due to prioritization of loop hoisting and generic pattern extraction— nonetheless it provides a structured basis for future symbolic equivalence and constraint tracking.

The Symbolic State

The `SymbolicState` structure accumulates symbolic reasoning across LLVM IR instructions. It maintains:

- `value_exprs`: Maps LLVM integer values to `SymExprs`.
- `instr_exprs`: Maps instructions to their derived expressions.

- `memory_accesses`: Tracks memory loads/stores with their access range.
- `checked_ranges`: Memorizes already instrumented address ranges.
- `assumptions`: Deduces value constraints from `icmp` comparisons.
- `mem_access_addr`: Maps loads/stores to symbolic addresses.
- `induction_vars`: Tracks loop induction patterns (e.g., `i in start..end`).

It is built by analyzing each instruction within a `BasicBlock`, extracting symbolic expressions and memory access metadata. During this pass, a dedicated handler processes each relevant instruction opcode:

- **Arithmetic and pointer arithmetic instructions** (`add`, `sub`, `mul`, `shl`, `getelementptr`, etc.) are symbolically evaluated and stored in `value_exprs` and `instr_exprs`.
- **Load/store operations** update memory access metadata. Loads are tracked with a unique provenance counter and associated symbolic address in `mem_access_addr`, while stores increment the `store_counter` and update `memory_accesses`.
- **Phi nodes** are parsed to identify loop induction patterns, inserting entries into `induction_vars`.
- **ICmp instructions** are analyzed to derive value constraints, which are accumulated in the `assumptions` map.
- **Function calls** are classified as safe or unsafe. Unsafe or unknown calls conservatively increment the store counter and mark potential memory growth.

Each `BasicBlock` receives its own independently constructed `SymbolicState`, which serves as a symbolic snapshot of its internal semantics propagated along the control-flow graph (CFG) using a reverse post-order traversal. This traversal ensures that predecessor states are merged into a block's state before that block is analyzed, preserving dominance relationships.

Merging is done conservatively: only information that is consistent across all incoming states is preserved and in OneOf if related to address. This conservative strategy is essential to soundness—it guarantees that any optimization decisions made (such as eliding a memory check) are valid across all possible execution paths. Despite this caution, a significant amount of useful information is retained due to the SSA form of LLVM IR.

Key fields, such as the `store_counter`, are merged using max-semantics to preserve upper bounds for check placement logic. Symbolic expressions for values and instructions are merged using a conservative symbolic join via `SymExpr::merge_conservative`, or wrapped in `OneOf` when ambiguity arises.

This propagation is further refined within loops using fixed-point analysis. Loop are iteratively reanalyzed using merged states from the loop body, allowing symbolic bounds (from induction variables and range assumptions) to stabilize. This process allows loop-aware optimizations that are both safe and effective in reducing redundant memory checks.

Listing 2: Symbolic value expression updates across loop iterations

```
Value Expressions:
    "%30 = load i64, ptr %29, align 4" =>
    OneOf([((([v0x5ac6587e11a0]%0 + v0x5ac6587ec3c8) +
         16)]%2, [(([v0x5ac6587e11a0]%0 +
         v0x5ac6587ec3c8) + 16)]%4)

    "%35 = load i64, ptr %34, align 4" =>
    [(([v0x5ac6587e11a0]%0 + v0x5ac6587ec3c8) + 24)]%3

    "%25 = load i64, ptr %24, align 4" =>
    OneOf([((([v0x5ac6587e11a0]%0 + v0x5ac6587ec3c8) +
         8)]%1, [(([v0x5ac6587e11a0]%0 +
         v0x5ac6587ec3c8) + 8)]%4)

    "%20 = load i64, ptr %19, align 4" =>
    [([v0x5ac6587e11a0]%0 + v0x5ac6587ec3c8)]%0
```

Tracking Memory Accesses and Store Counter

Memory tracking in `SymbolicState` relies on two core mechanisms: the `memory_accesses` map, which records symbolic load and store expressions along with their access ranges, and the `store_counter`, a global monotonic counter used to tag the provenance of `load` expressions.

Each `load` is tagged with the current value of the `store_counter` when it is first encountered via the `add_load_memory_access` method. This counter is incremented on any `store` instruction or on encountering function calls that may alter memory state. The purpose is to conservatively distinguish between loads made before and after potentially interfering writes. However, if the memory region accessed by the function is known and safe, or if the call is annotated as harmless (e.g., intrinsic or known allocation routines), the counter is not updated.

In the `handle_call` function, calls to safe functions (such as `mem_check` or any function explicitly marked as non-interfering) are ignored. For potentially unsafe calls (unknown)`memory_accesses` is cleared or pruned depending on whether the base pointer is retained. This ensures that all tracked load/store metadata reflects memory that has not been overwritten or modified indirectly.

For `store` instructions, the `add_store_memory_access` method scans all previously recorded accesses and removes any whose address range may overlap with the store. This operation is guided by symbolic disjointness checks, using `is_disjoint_exact`. If the memory access is outside the currently tracked base pointer, or if the overlap cannot be ruled out, the corresponding entries are invalidated.

The `store_counter` enables fine-grained tracking of memory modification timing. By associating loads with the counter state at the point of their insertion, and updating the counter only when unsafe or overlapping memory writes occur, the symbolic system can conservatively preserve memory access knowledge across safe paths—especially valuable in loop analysis and inter-block optimizations.

Loop-Aware Optimization

Loops are detected using dominator tree analysis and control-flow graph (CFG) back edges[2]. Each natural loop is characterized by a header and a tail (a back edge source), from which the full loop body is collected.

Within loops, Symbolic states are refined through fixed-point iteration over the loop body. This iterative process ensures a stable symbolic summary of memory access behavior under loop-induced transformations.

Memory accesses in loops are summarized in a structure called `LoopMemoryContext`. This context encapsulates:

- Induction variables and their start, step, and update patterns
- Symbolic memory access expressions tied to induction
- Derived symbolic bounds for access ranges
- Canonical symbolic range and step size expression for total memory footprint

The symbolic step is inferred using the `LoopStepKind` abstraction (`Add`, `Sub`, `Mul`), with helper methods to deduce concrete steps when possible. This enables both algebraic manipulation and concrete estimation of loop iterations.

For example, consider the loop:

```
for (int i = 0; i <= 10; ++i) {
load(base + i * 8);
}
```

Symbolically, this is represented by a start at `base`, a step of `8`, and a bound at `base + 80`. The symbolic engine identifies `i` as the loop induction variable, extracts the constant bound from the `ICmp` comparison, and models the address expression `base + i * 8`.

The optimizer emits a single memory check before loop entry that guards the entire access region: `check(base, base + 80)`.

If the stride or iteration bound is symbolic or only partially known, a conservative symbolic range is constructed using a fallback expression of the form: `check(base, base + (bound - start) * step)`.

However, it is often the case that the induction variable governing the loop's stop condition differs from the one used directly in the memory access expressions. To correctly derive a safe check range in such cases, the analysis first computes the number of loop iterations based on the induction variable controlling the loop exit. Then, using the symbolic access patterns associated with memory operations, it determines the maximum and minimum symbolic offset contributed by the memory-related induction variable(s). The final range is computed by multiplying the total number of iterations by this maximum/minimum symbolic stride, and adding it to the base expression. This enables the derivation of a conservative but tight upper and lower bound even in the presence of multiple, potentially disjoint induction variables.

This loop-aware strategy significantly reduces redundant runtime checks, particularly in nested or long-running loops, by validating access ranges statically.

Memory Access Modeling and Grouping

Each `load` or `store` is analyzed to extract a symbolic address expression. These addresses are tracked per instruction and used to form *access pattern groups*, which identify recurring memory patterns such as loop-strided accesses.

For instance, all accesses of the form `base + i * 8 + k` for fixed `k` within a loop body are grouped together. These groups enable emitting a single bounds check for the entire range instead of per-instruction checks.

Grouping is applied both:

- **Intra-block**: Within individual basic blocks.
- **Inter-block**: Across loop boundaries, inserted at pre-headers.

Memory Check Insertion Strategy

Memory check insertion is governed by configurable compilation modes, controlled via feature flags during the `cargo build` process. These modes represent increasing levels of sophistication in symbolic reasoning and check elimination, corresponding to evaluation variants:

- **No Emission (base)**: No memory checks are inserted. This baseline variant serves to isolate transformation overhead without safety enforcement.

- **Naive (check)**: A direct bounds check is emitted for every `load/store` instruction. This serves as a correctness baseline and reflects typical runtime-enforced sandboxing.
- **Redundancy-Aware Naive (opt1)**: This variant extends the naive mode by avoiding repeated checks on memory addresses that have already been validated in the current symbolic context. Symbolic expressions are matched using `SymExpr` representations and deduplicated via the `checked_sym_exprs` registry in the symbolic state.
- **Grouped Intra-block + Naive Inter-block (opt2)**: In this configuration, intra-block memory accesses are grouped by symbolic similarity and only one representative check is emitted per group. For loop-related access patterns, checks are hoisted to loop headers using induction-bound analysis. Inter-block accesses not related to induction variables fall back to naive checking within their respective blocks.
- **Fully Optimized Grouped Emission (opt3)**: In the context of loop optimization, `opt3` attempts to treat the entire loop body—even when spanning multiple basic blocks—as a single unit. It hoists all memory checks, including those not directly tied to induction variables, to the loop header by statically resolving their safety through symbolic reasoning. If symbolic constraints are insufficient to prove the bounds soundly, the transformation fails at compile time to preserve safety—no fallback is inserted. The intended future direction is to switch dynamically to a safer strategy, akin to `opt2`, where non-induction related accesses within the loop fall back to naive checks for the remainder of the loop execution. However, this runtime recovery path was not implemented in the current prototype, meaning that symbolic failure in `opt3` results in compilation failure rather than a soft fallback.

Optimizations rely on canonicalized symbolic address expressions and precise grouping of memory accesses. Key insertion strategies include:

- **Loop-aware Inter-block Checks**: Memory access patterns referencing loop induction variables are analyzed through the `LoopMemoryContext` structure. These patterns are symbolically summarized to capture the full range of accessed memory addresses across all iterations. A single memory check for min and max value is inserted before the loop (typically in a pre-header block) to validate the entire access range. This avoids per-iteration checks while maintaining safety.

Listing 3: Inter-block memory check inserted before loop

```
mem_check_block: ; preds = %3
%base = load ptr, ptr %1, align 8
%base_plus_offset = getelementptr i8, ptr %base,
     i64 %0
%base_plus_offset1 = getelementptr i8, ptr %
     base_plus_offset, i64 824
call void @mem_check(ptr %1, ptr %2, ptr %
     base_plus_offset1)
call void @mem_check(ptr %1, ptr %2, ptr %
     base_plus_offset)
br label %10
```

This example demonstrates a typical pre-header insertion for a loop ranging from `i = first argument of the function` to `i < 100`, where the maximum symbolic offset is statically known to be 824 bytes (800 from iteration steps and 24 from unrolling effects).

Note: As checks are inserted at the start of the pre-header block, any address computation or induction value required must be hoisted or recomputed at this location. This may involve instruction duplication or transformation if the needed values are not yet in scope.

- **Intra-block Grouped Checks**: Within a single basic block, memory accesses are grouped according to their symbolic address pattern. Each group emits a single memory check (min/max) for the combined address range. This strategy benefits from LLVM's SSA form, where variable identities are stable within a block, allowing precise deduplication and aggressive minimization of redundant checks. However, memory check insertion must respect instruction dependencies—if a check relies on values produced by earlier instructions (e.g., loads), it cannot be placed at the block's entry but must follow those instructions. Conversely, when the required values are not themselves memory-dependent within the same block, these computations can be safely relocated to the beginning of the block, enabling early check insertion. checks.
- **Fallback Avoidance**: In `opt3` mode, fallback to naive check insertion is explicitly disabled. Only access patterns that are provably safe using symbolic analysis are allowed. If no safe symbolic range can be established, the transformation fails at compile time. This ensures early detection of unsupported patterns and enforces a safety-by-construction principle during development and evaluation.

This stratified check insertion strategy was designed specifically for evaluation purposes, allowing controlled comparison of performance, safety, and transformation robustness across progressively optimized modes.

Evaluation

Benchmark Structure and Methodology

To evaluate the performance impact of our symbolic sandboxing pipeline, we compiled and executed a suite

Table 1: Benchmarking results showing average execution times (in microseconds) for each implementation variant. The table includes absolute runtimes (mean ± standard deviation) and relative speedups compared to the baseline `check` implementation.

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%

of representative benchmarks under multiple optimization configurations. Each benchmark was run in a simulated multi-contract environment, where multiple *.so* modules were loaded sequentially in random order to emulate cache effects and scheduling variability. To explore sensitivity to input scale and runtime conditions, we used two batch sizes of 65 536 and 32 464 executions for each benchmark.

A detailed description of the visualization format and per-benchmark analysis is provided in *Appendix* .

However, the complexity of the test kernels was constrained by current limitations of the Droplet toolchain. The current SMIR layer supports only a minimal subset of operations—many data types such as `f32` and `f64` currently only implement constant values. As a result, the benchmarks focus on integer-based arithmetic and control-flow patterns that exercise memory access and sandboxing logic, while avoiding unsupported features at the IR level.

The test suite emphasizes *loop-centric computations*, as the most impactful optimizations in this work target *inter- and intra-block memory check elimination* within fixed point constructs. Additionally, a range of tests was conducted to verify the **correctness** and **coverage** of memory safety checks. Although memory safety is enforced at the granularity of `WebAssembly` pages across the entire smart contract address space, our experiments confirm that any access performed *outside those bounds* was consistently and reliably detected by the inserted checks.

In some of these correctness-focused tests, the system—being a research prototype developed under time constraints—occasionally failed to compute valid bounds for merging access groups due to unhandled symbolic edge cases. While a fallback to naive checking would be appropriate in a production-grade pipeline, the current implementation aborts compilation upon such errors. As a result, a small number of tests could not be compiled successfully and thus do not appear in the reported performance results.

Setup

All benchmarks were executed on a laptop running Arch Linux with kernel version `6.14.6-arch1-1`. The machine is equipped with a 12th Gen Intel® Core™ i7-1265U CPU, 15 GiB of RAM, and a 953.9 GB NVMe solid-state drive (PC801). Compilation was performed using `gcc` version 15.1.1 (20250425) and `clang` version 19.1.7. Although the system includes an integrated GPU (reported as `02.0 VGA compatible controller`), all benchmark execution and compilation were performed on the CPU. This configuration reflects the hardware constraints of a student research environment and may not represent high-end server performance.

Question

What is the performance cost of introducing symbolic memory safety instrumentation in WebAssembly-based smart contract runtimes, and how effectively can symbolic compilation optimizations mitigate this cost? Specifically, we aim to quantify the overhead of **naive memory checks** and measure the performance gains from **progressively applied optimizations** across a range of computational patterns.

Observation

Across all benchmarks, *naive memory sandboxing* introduces a substantial runtime overhead relative to the no-check baseline—ranging from **1.5× to over 10×**. For instance, the `addbounded` benchmark increases from 2.75 μs to 29.04 μs under naive checking. However, successive application of optimizations (`Opt1`–`Opt3`) yields notable reductions. On average, `Opt1` recovers **30–50%** of the overhead, while `Opt2` and `Opt3` often surpass **70–80%** overhead reduction, with some benchmarks nearly matching the no-check baseline (e.g., `add1` and `reverse`). *Variance also tends to decrease with deeper optimization.*

There are exceptions. Benchmarks like `conditional`, `fibonaccilike`, and `slidewindow` show limited or inconsistent speedups in `Opt1`, with more substantial gains only appearing in later stages (`Opt2`/`Opt3`) or not at all. Some tests (e.g., `matrix`, `fibonaccilike`) show high baseline variance due to data-dependent execution paths.

Deduction

The overhead introduced by naive sandboxing stems primarily from *repeated dynamic memory bounds checks*, which dominate small kernels with tight loops or many accesses (`addbounded`, `nested`, `redundant`). The initial optimization stage (`Opt1`) typically *eliminates redundancy* and *coalesces checks* within individual blocks, which explains the consistent **30–50%** gain across benchmarks.

`Opt2` improves upon this by performing *inter-block loop-aware coalescing*, allowing checks across iterations or across multiple control-flow paths to be merged, particularly effective in benchmarks with structured control flow and predictable memory access patterns (`2d`, `matrix`, `nested`).

`Opt3` enhances this by *restructuring basic blocks* to consolidate memory checks at the beginning of each block and by *merging access pattern groups* when safe. This avoids scattered or repeated checks and is especially impactful in tight control-flow kernels where multiple accesses follow similar bounds (`reverse`, `redundant`, `stride`). In contrast, benchmarks with dynamic branching or highly data-dependent access patterns (e.g., `conditional`, `fibonaccilike`) resist full optimization, as conservative analysis prevents safe merging or relocation of checks.

Conclusion

Symbolic memory sandboxing introduces **measurable overhead**, but our pipeline significantly reduces its impact. For most benchmarks, `Opt2` and `Opt3` recover **60–80%** of the naive overhead, often reaching near-baseline performance. The approach is especially effective on *regular access patterns*, *predictable control flow*, and *loop-dominated workloads*. However, optimization is less effective in *data-dependent or branching-heavy kernels*, where static reasoning is harder.

These results validate the **practical viability** of compile-time symbolic sandboxing for WebAssembly smart contract runtimes, striking a strong balance between *safety and efficiency*, especially when higher-level symbolic reasoning is applied.

Future Work: Integrating SymExpr with Sea of Nodes

While this project focused on symbolic sandboxing within Droplet's SSA-based LLVM IR pipeline, a natural extension would be to introduce a Sea of Nodes (SoN) intermediate representation. SoN structures programs as graphs of operations, enabling powerful global optimizations such as value numbering, loop-invariant code motion, and precise dataflow tracking.

A viable integration strategy would preserve `SymExpr` as the core semantic layer while using SoN as the structural IR. Each SoN node would carry an optional symbolic payload:

Listing 4: Proposed SoN node design

```
pub struct SonNode {
pub id: NodeId,
pub opcode: SonOp,
pub inputs: Vec<NodeId>,
pub symexpr: Option<SymExpr>,
}
```

This design cleanly separates concerns: SoN captures control, data, and memory dependencies; `SymExpr` expresses semantics for reasoning, equivalence checking, and memory range inference.

In particular, memory check inference could operate directly on symbolic payloads of `Load`/`Store` nodes, using the existing `ValueRange` framework for bounds validation. Control constructs like `If`, `Phi`, and `Region` would remain in SoN, decoupled from symbolic logic, preserving SSA-aware flow while enabling future global optimizations.

This modular layering would make symbolic checks more robust and expressive while paving the way for further optimization passes within Droplet's architecture.

Conclusion

This project set out to explore the feasibility of compile-time symbolic memory sandboxing for WebAssembly-based smart contract execution. Motivated by the high safety and determinism demands of replicated state machines, we developed a novel LLVM-based pipeline that statically emits memory bounds checks. Through the design and implementation of a symbolic algebra (`SymExpr`), along with control-flow and loop-aware analysis, we demonstrated that significant runtime overhead from naive memory sandboxing can be mitigated—achieving up to 80% performance recovery in representative workloads.

While the scope of this project was ambitious, it served as a valuable exploration of several promising research directions. The initial aim to integrate Sea of Nodes representations, enhance symbolic equivalence tracking, and implement comprehensive assumption reasoning laid a strong conceptual foundation, even if not all components could be fully realized within the project timeline. In retrospect, the breadth of the endeavor may have stretched the available development time, but it also enriched the overall design and opened clear pathways for future work.

This work yielded meaningful contributions: the symbolic engine showed real speedup in generic loop-heavy code, hoisting checks in nested control flows, and enabling memory grouping strategies previously unseen in the Droplet toolchain. These advances provide a solid foundation for future efforts in symbolic reasoning and static analysis for smart contract safety.

While not all planned components reached full maturity, the core contribution—loop-aware symbolic bounds checking—demonstrated strong practical value, and its successful integration into a working compilation pipeline confirms the viability of symbolic sandboxing for safe, efficient smart contract execution.

Acknowledgements

I would like to sincerely thank Gauthier Voron for his guidance, insightful feedback, and for authoring the original implementation of *Droplet*, which laid the foundation for this work. His support throughout the project has been deeply appreciated.

I am also grateful to Professor Rachid Guerraoui for his role as our academic supervisor and for his courses on distributed and concurrent systems, which greatly informed the design.

Special thanks go to David Schroeter for his close collaboration throughout the development of this work. We worked in parallel on complementary aspects of the runtime infrastructure, continuously supporting each other to ensure design coherence and compatibility across our respective contributions.

ChatGPT was used to assist with phrasing during the writing of this report, in accordance with EPFL guidelines. All content was reviewed to ensure accuracy and academic integrity.

References

1. Dirren EA. Efficient Time Sandboxing for State Machine Replication-oriented Compilers. Technical Report. Supervised by Gauthier Vorona. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne (EPFL), 2025 Jan. Available from: mailto:elija-angelo.dirren@epfl.ch
2. Andriesse D. Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. Print Book and FREE Ebook available. San Francisco, CA: No Starch Press, 2018 Dec. Available from: https://practicalbinaryanalysis.com
3. Click C and Cooper KD. Combining analyses, combining optimizations. ACM Trans. Program. Lang. Syst. 1995 Mar; 17:181–96. DOI: 10 . 1145 / 201059 . 201061. Available from: https://doi.org/10.1145/201059.201061
4. Click C. From Quads to Graphs: An Intermediate Representation's Journey. 1997 Feb
5. Click C and Paleczny M. A simple graph-based intermediate representation. SIGPLAN Not. 1995 Mar; 30:35–49. DOI: 10.1145/202530.202534. Available from: https://doi.org/10.1145/202530.202534
6. Click C and Paleczny M. A simple graph-based intermediate representation. *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. IR '95. San Francisco, California, USA: Association for Computing Machinery, 1995 :35–49. DOI: 10.1145/202529.202534. Available from: https://doi.org/10.1145/202529.202534
7. Muchnick S. Advanced Compiler Design and Implementation. 1st. Includes case studies from SPARC, POWER, Alpha, and Pentium compilers. San Francisco, CA: Morgan Kaufmann, 1997 Aug
8. Schinz M. CS-420: Advanced Compiler Construction. https : / / cs420 . epfl . ch/. Course material, EPFL. Covers compiler design for functional and object-oriented languages, including IRs, optimizations, and runtime systems. 2024
9. Schroeter D. Compiler-Based Microsection Scheduling for Parallel Smart Contract Execution. 2025. Available from: mailto:david.schroeter@epfl.ch

Appendix

Per-Benchmark Visualization and Analysis

Each benchmark in this appendix follows a consistent structure to aid in interpreting performance and optimization effects:

- **Pseudocode (left):** A concise algorithmic summary of the test kernel, illustrating memory access and control flow patterns critical for symbolic analysis.
- **Execution trace (right):** A runtime profile displaying execution time distributions and batch-level variance. Benchmarks are executed repeatedly with randomly ordered `.so` module loads to simulate cache effects and interference akin to multi-contract execution.
- **Optimization comparison (bottom):** Execution times under different sandboxing and optimization configurations, revealing both overhead and performance improvements.

Two batch sizes of 65 536 and 32 464 are used to evaluate sensitivity to input scale and runtime variability.

Algorithm 1: 2d.c — Increment square matrix elements

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Let `dim` ← $\lfloor\sqrt{\texttt{nb\_elem}}\rfloor$;
Initialize `b` ← 0;

for $i \leftarrow 0$ **to** $dim - 1$ **do**
  for $j \leftarrow 0$ **to** $dim - 1$ **do**
    `ptr`[$i \cdot$ `dim` $+ j$] ← `ptr`[$i \cdot$ `dim` $+ j$] $+1$;

return 0;

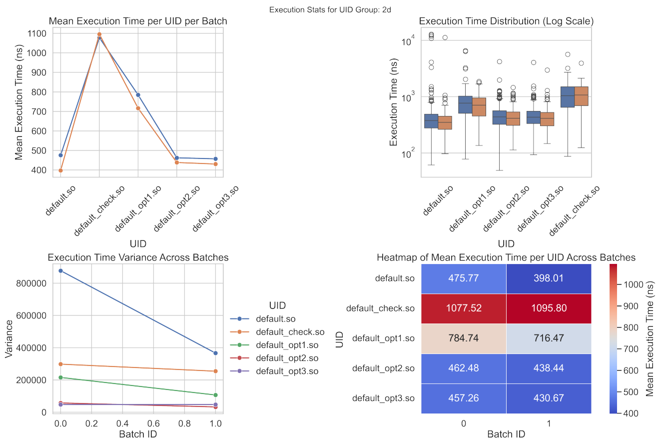
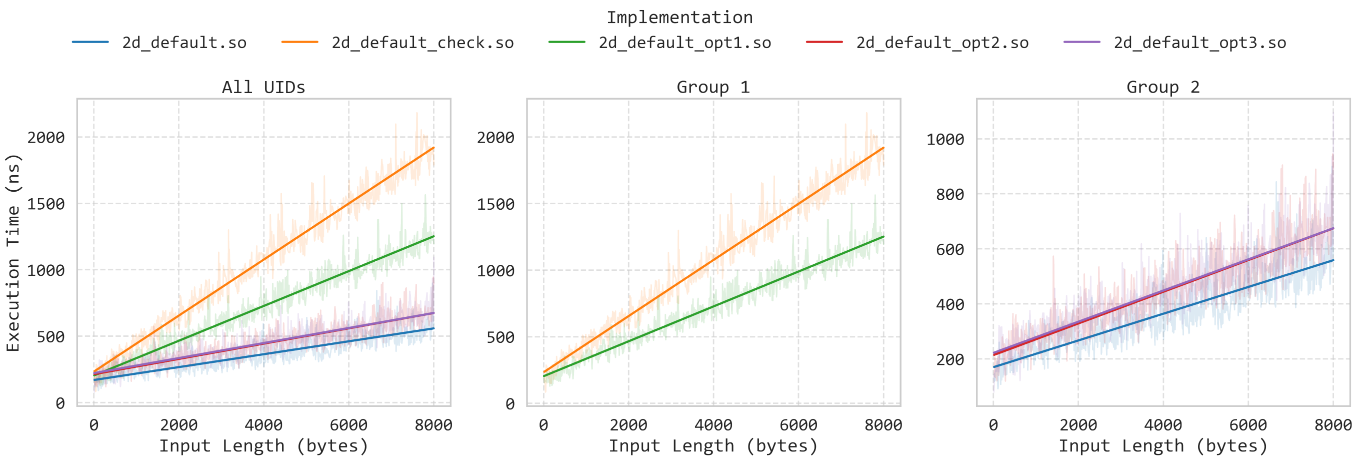


Figure 3: Execution trace visualization of `2d.c`



Algorithm 2: add1.c — Increment each array element

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;

for $a \leftarrow 0$ **to** $nb\_elem - 1$ **do**
    `ptr[` a `]` ← `ptr[` a `]` +1;

return 0;

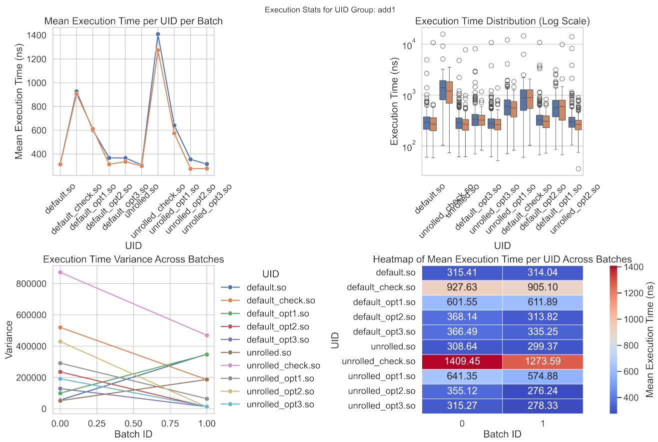
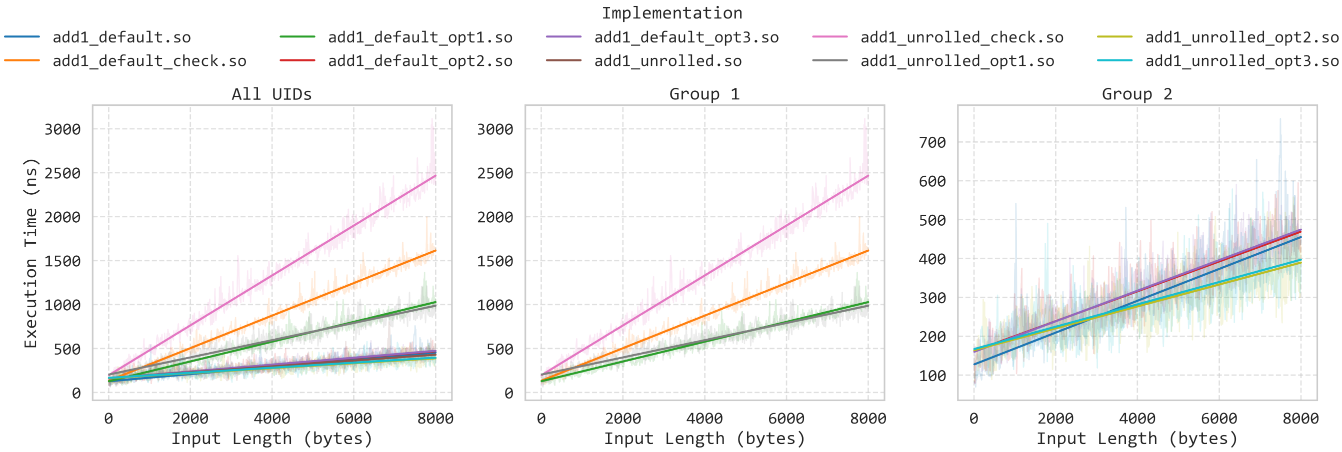


Figure 4: Execution trace visualization of `add1.c`



Algorithm 3: addbounded.c — Bounded increments and nested accumulation

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns the accumulated result of nested increments

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;

for $a \leftarrow 150$ **to** 1 **by** -1 **do**
    `ptr[` a `]` ← `ptr[` a `]` +2;
    `b` ← `b` + `fun(`ptr`)`;

return `b`;

Function fun(ptr):
    Initialize `tmp` ← 0;
    for $i \leftarrow 0$ **to** 99 **do**
        `ptr[` i `]` ← `ptr[` i `]` +1;
        `tmp` ← `tmp` + `ptr[` i `]`;
    return `tmp`;

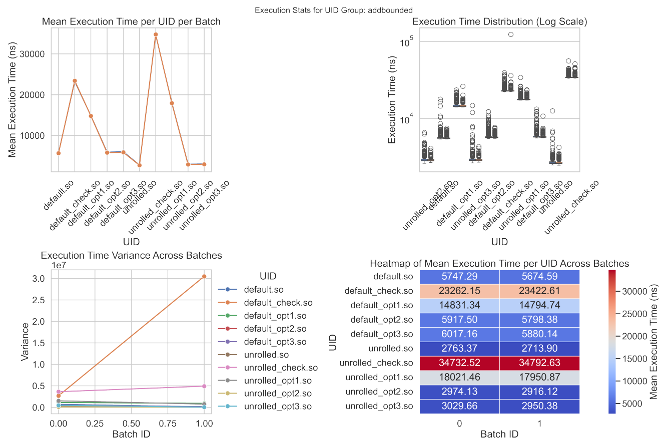
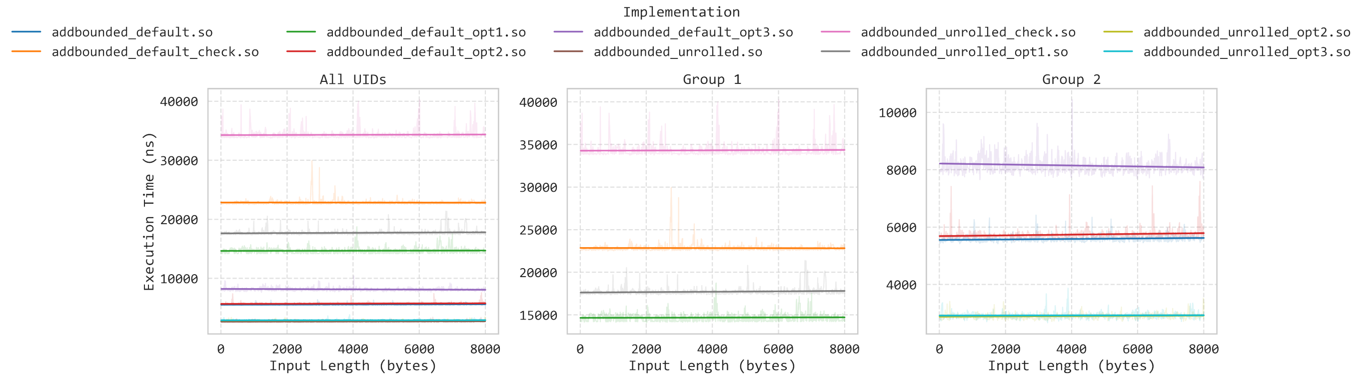


Figure 5: Execution trace visualization of `addbounded.c`



Algorithm 4: conditional.c — Increment only even elements

Input : data – a pointer to a memory buffer
size – the total size (in bytes) of the buffer

Output: Returns 0

Let ptr ← reinterpret data as array of 64-bit integers;

Let nb_elem ← size / sizeof(uint64_t);

Initialize b ← 0;

for $a \leftarrow 0$ **to** $nb\_elem - 1$ **do**
  if *ptr[a] mod 2 = 0* **then**
    ptr[a] ← ptr[a] +1;

return 0;

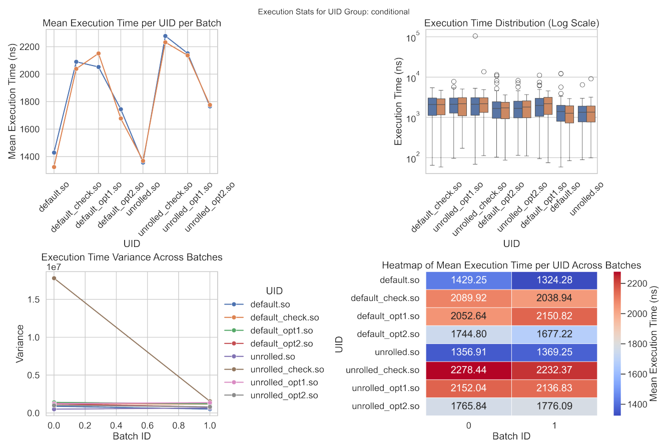
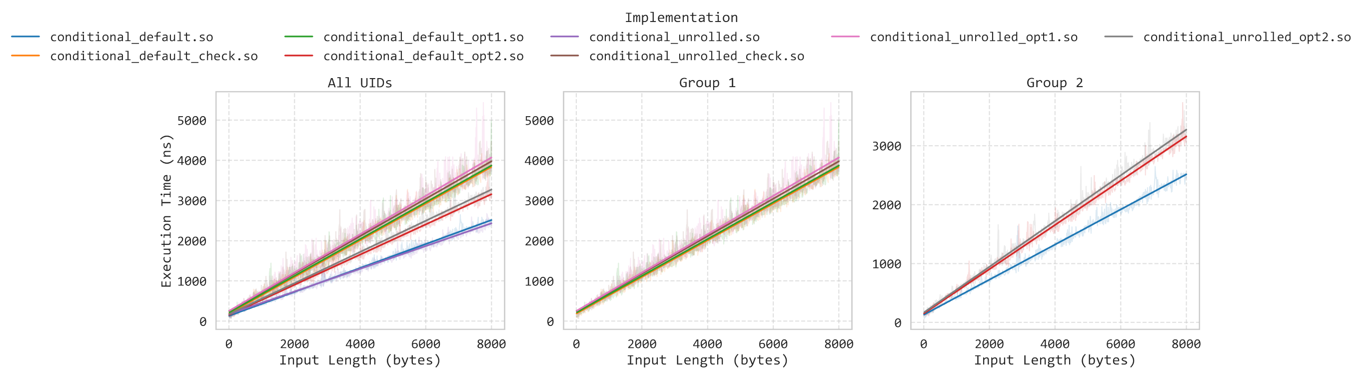


Figure 6: Execution trace visualization of `conditional.c`



Algorithm 5: fibonaccilike.c — Fill buffer with Fibonacci-like values

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns the last value in the buffer

Let `buf` $\leftarrow$ reinterpret `data` as array of 64-bit integers;
Let `n` $\leftarrow$ `size` / sizeof(uint64_t);
if $n < 3$ **then**
  return 0;
for $i \leftarrow 2$ **to** $n - 1$ **do**
  `buf` [i] $\leftarrow$ `buf` [$i - 1$] $+$ `buf` [$i - 2$];
return `buf` [`n` $- 1$];

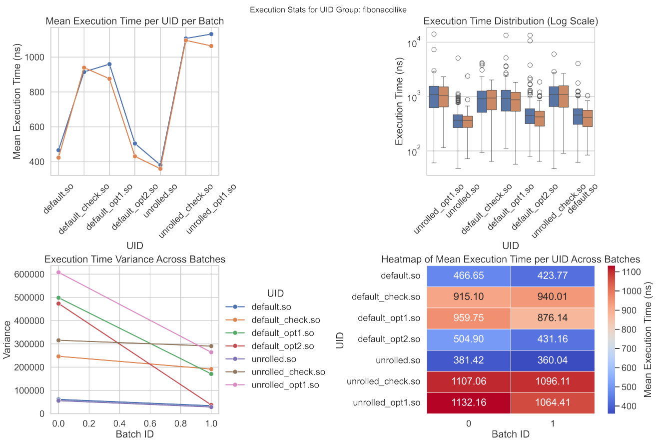
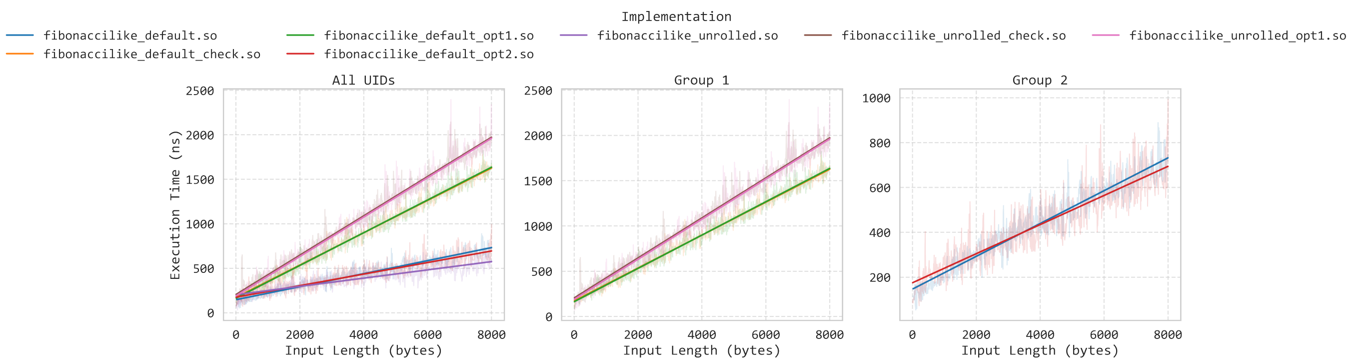


Figure 7: Execution trace visualization of `fibonaccilike.c`



Algorithm 6: matrix.c — Multiply two square matrices (A × B → C)

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns `C[0]` after matrix multiplication

Let `buffer` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Let `n` ← 1;
while $n \cdot n \cdot 3 \leq$ `nb_elem` **do**
  $n \leftarrow n + 1$;
$n \leftarrow n - 1$;
if $n = 0$ **then**
  return 0;

Let `A` ← `buffer`;
Let `B` ← `buffer` $+ n \cdot n$;
Let `C` ← `buffer` $+ 2 \cdot n \cdot n$;

for $i \leftarrow 0$ **to** $n - 1$ **do**
  for $j \leftarrow 0$ **to** $n - 1$ **do**
    Initialize `sum` ← 0;
    for $k \leftarrow 0$ **to** $n - 1$ **do**
      `sum` ← `sum` + `A[` $i \cdot n + k$ `]` · `B[` $k \cdot n + j$ `]`;
    `C[` $i \cdot n + j$ `]` ← `sum`;

return `C[0]`;

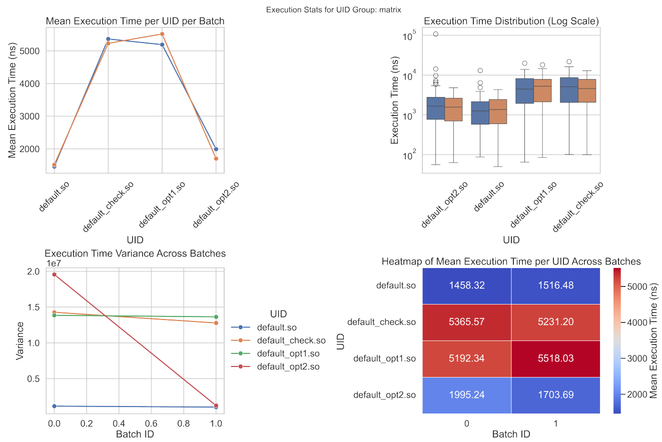
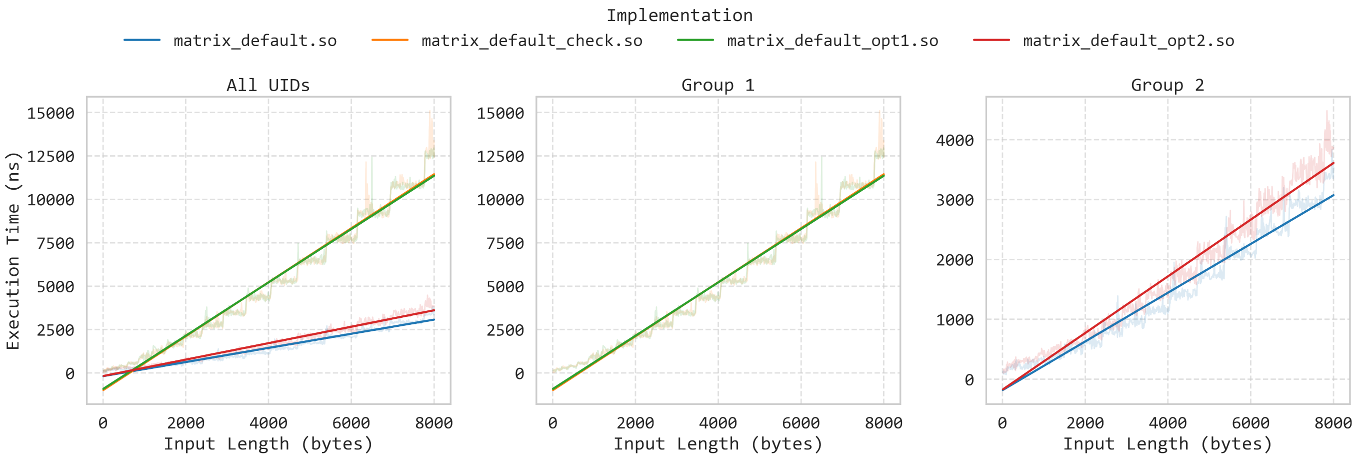


Figure 8: Execution trace visualization of `matrix.c`



Algorithm 7: nested.c — Increment elements in blocks of 4

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;

for $i \leftarrow 0$ **to** $nb\_elem/4 - 1$ **do**
  for $j \leftarrow 0$ **to** 3 **do**
    `ptr`[$i \cdot 4 + j$] ← `ptr`[$i \cdot 4 + j$] +1;

return 0;

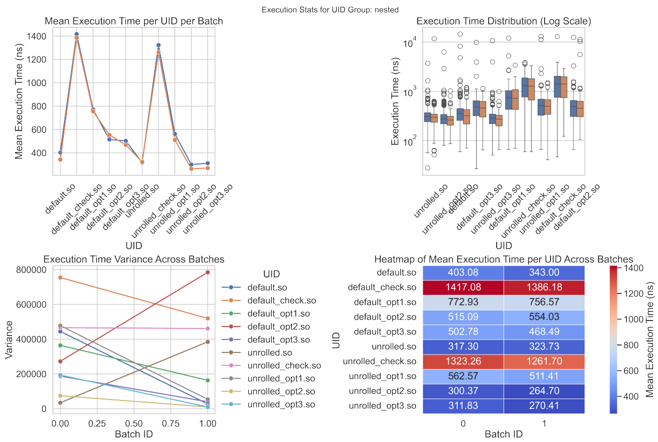
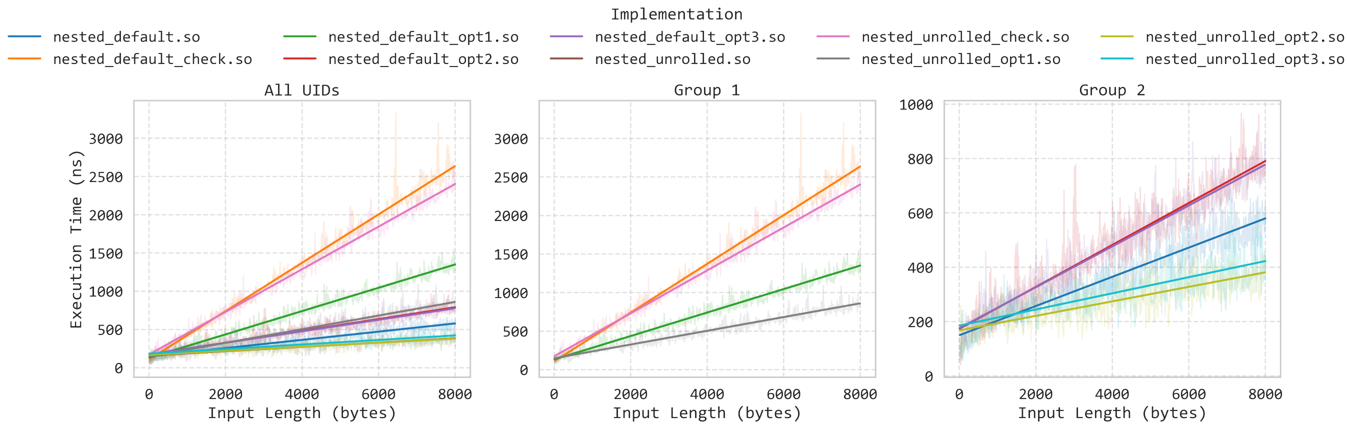


Figure 9: Execution trace visualization of `nested.c`



Algorithm 8: prefix.c — In-place prefix product computation

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns the last element of the modified buffer

Let `buf` ← reinterpret `data` as array of 64-bit integers;
Let `n` ← `size` / sizeof(uint64_t);
if $n = 0$ **then**
  return 0;

for $i \leftarrow 1$ **to** $n - 1$ **do**
  `buf`[i] ← `buf`[i] · `buf`[$i - 1$];

return `buf`[n − 1];

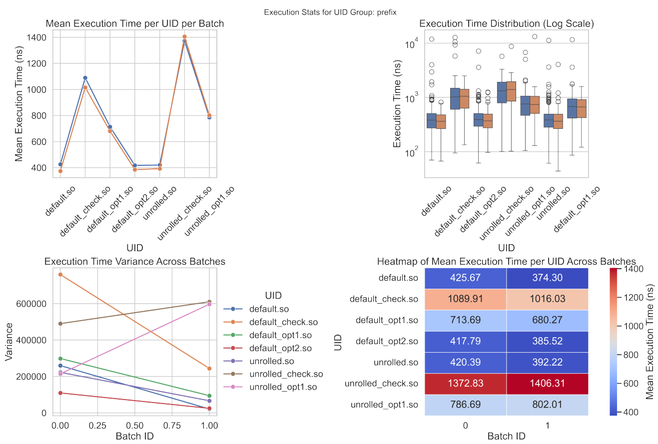
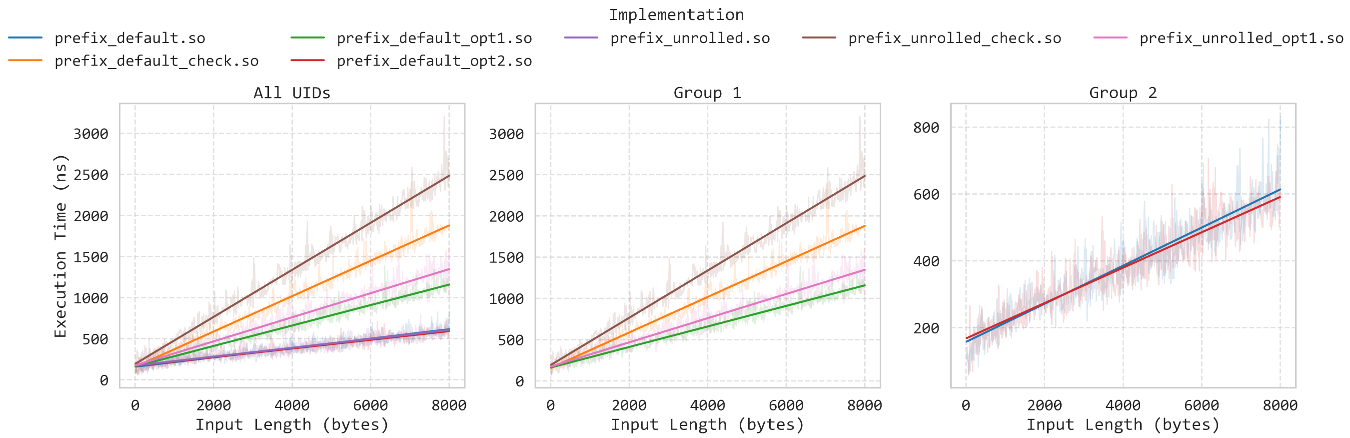


Figure 10: Execution trace visualization of `prefix.c`



Algorithm 9: redundant.c — Redundant bounds check during iteration

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;

for $a \leftarrow 0$ **to** $nb\_elem - 1$ **do**
  if $a < nb\_elem$ **then**
    `ptr[` a `]` ← `ptr[` a `]` +1;

return 0;

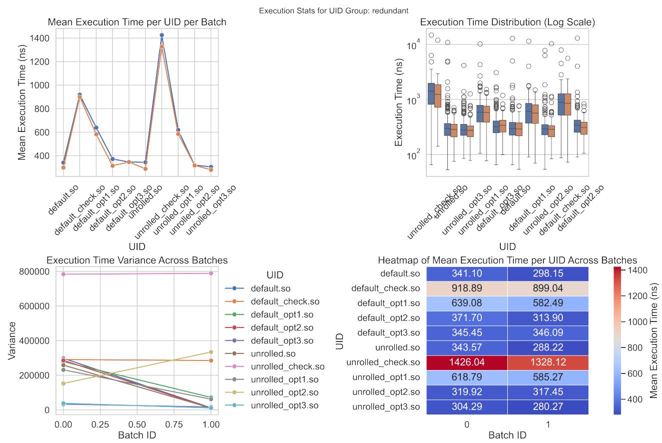
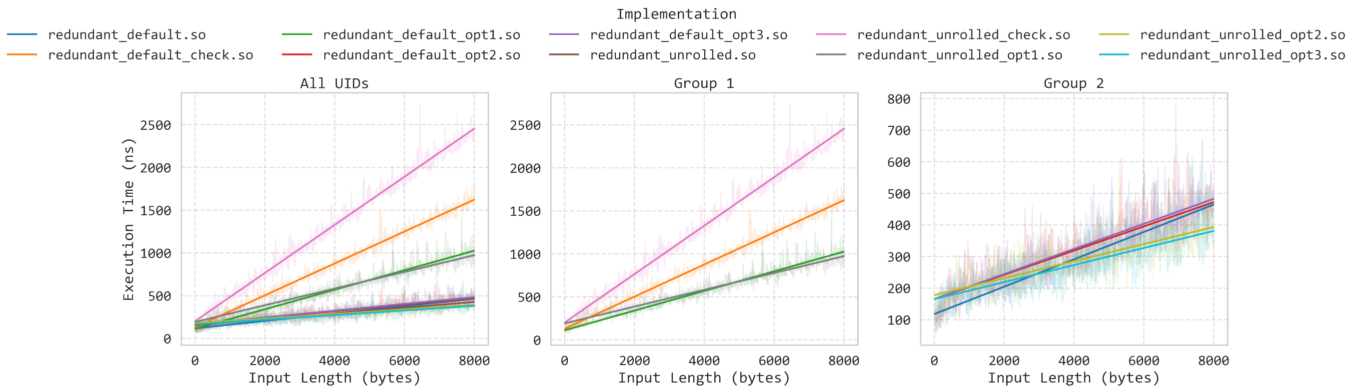


Figure 11: Execution trace visualization of `redundant.c`



Algorithm 10: reverse.c — Reverse-order element increment

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;
for $a \leftarrow$ *nb_elem* $- 1$ **to** 0 **by** -1 **do**
    `ptr`[a] ← `ptr`[a] +1;
return 0;

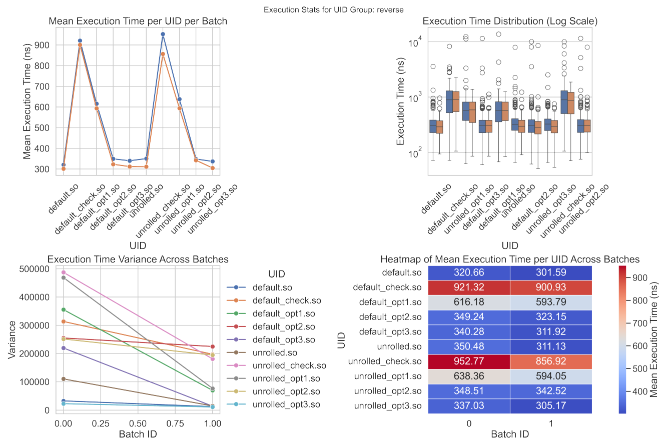
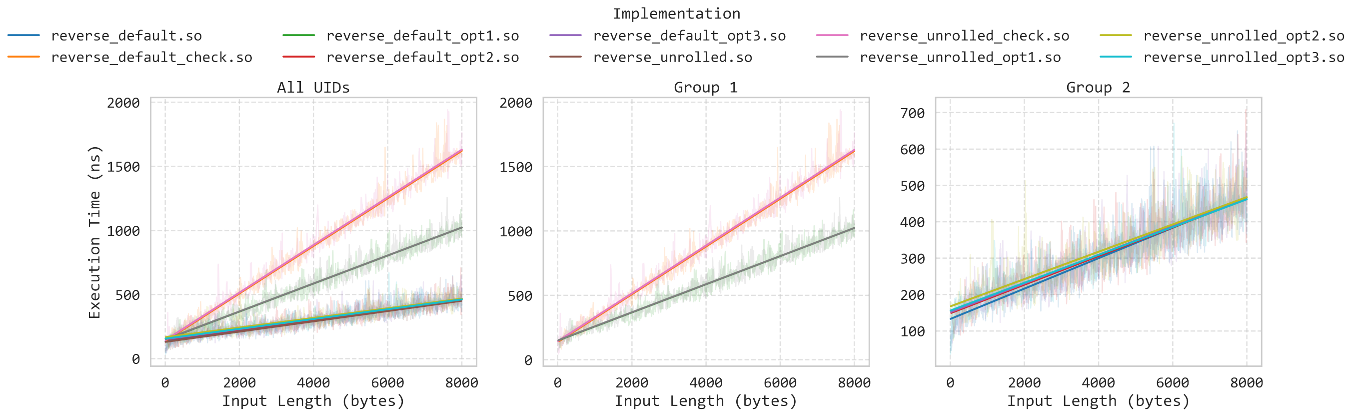


Figure 12: Execution trace visualization of `reverse.c`



Algorithm 11: slidewindow.c — Sliding window average of 5 elements

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns the first element of the modified buffer

Let `buf` ← reinterpret `data` as array of 64-bit integers;
Let `n` ← `size` / sizeof(uint64_t);
if $n < 5$ **then**
    return 0;
for $i \leftarrow 0$ **to** $n - 5$ **do**
    `buf`[i] ← (`buf`[i] + `buf`[$i+1$] + `buf`[$i+2$] + `buf`[$i+3$] + `buf`[$i+4$]) / 5;
return `buf`[0];

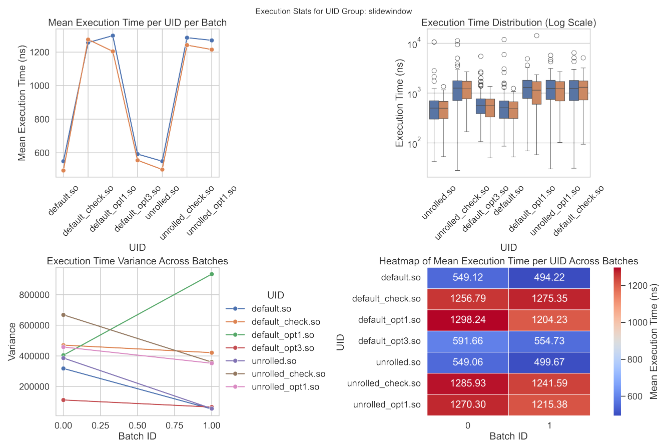
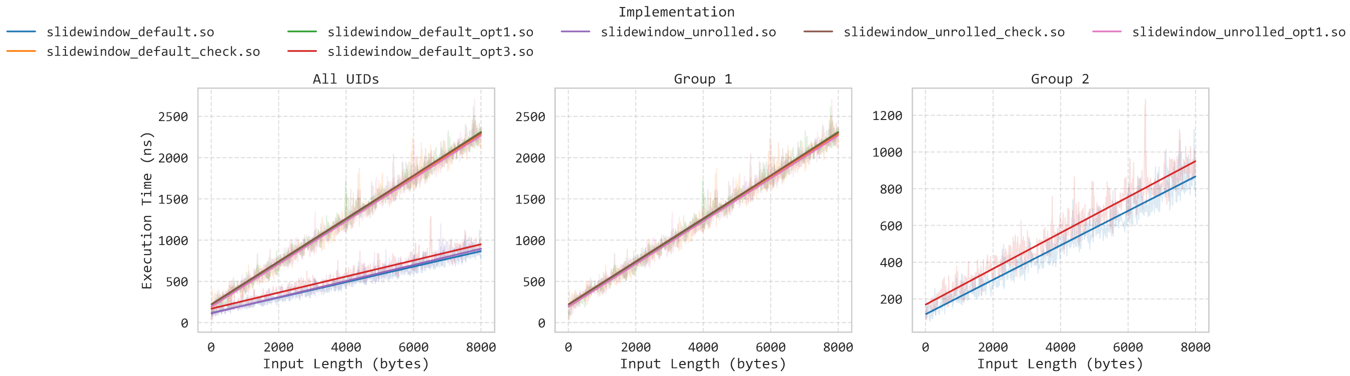


Figure 13: Execution trace visualization of `slidewindow.c`



Algorithm 12: stride.c — Increment every other element (stride 2)

Input : `data` – a pointer to a memory buffer
`size` – the total size (in bytes) of the buffer

Output: Returns 0

Let `ptr` ← reinterpret `data` as array of 64-bit integers;
Let `nb_elem` ← `size` / sizeof(uint64_t);
Initialize `b` ← 0;

for $a \leftarrow 0$ **to** $nb\_elem - 1$ **by** 2 **do**
  `ptr[` a `]` ← `ptr[` a `]` +1;

return 0;

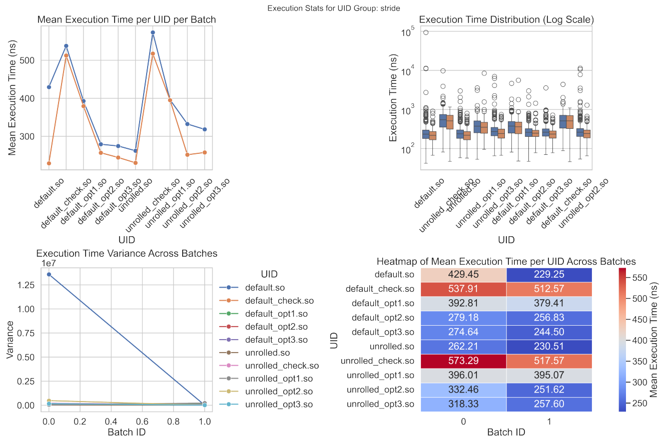
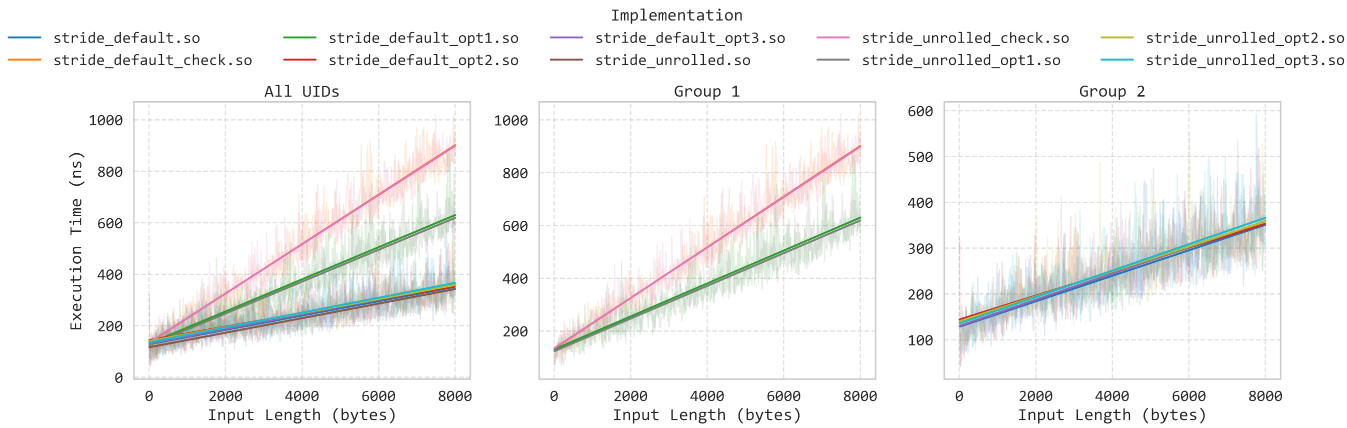


Figure 14: Execution trace visualization of `stride.c`



Smart Contract template

To ensure compatibility with *Drizzle* and *Droplet*, all benchmarks were adapted to a standardized C template. This interface, largely designed and implemented by David Schroeter as part of his foundational work "*Compiler-Based Microsection Scheduling for Parallel Smart Contract Execution*"[9], provides essential entry points and memory hooks for *Drizzle* execution. I am deeply grateful for David's contributions and ongoing collaboration, which have been critical in enabling and maintaining integration with our symbolic analysis pipeline.

The wrapper defines a fixed-size linear memory buffer (`wrapper_memory`) and exposes its base address via `get_wrapper_memory_addr()`, enabling *Droplet* and *Drizzle* to locate and structure memory in a concurrent environment.
It also includes exported allocation and deallocation routines (`alloc` and `dealloc`) for allocation logic. The primary contract logic is implemented in `droplet_entry()`, with an optional installation phase in `droplet_install()` useful for preparing the smart contract.
Additional hooks like `dump()` and `entry_ret_u64()` support contract state inspection and automated testing.

These wrapper functions define the contract between user-level smart contract code and environment, allowing the entire pipeline to operate robustly across a range of test cases while preserving compatibility with Drizzle.

Listing 1: Template of contract implementation in C

```c
#include <stdint.h>
#include <stddef.h>

#define WASM_MEMORY_SIZE 65536

#define WASM_EXPORT(name) \
  __attribute__((export_name(#name))) \
  name

static char wrapper_memory[WASM_MEMORY_SIZE] = { 0 };

// That function is needed only so that droplet can know where the beginning of the wrapper memory is.
void* WASM_EXPORT(get_wrapper_memory_addr)() {
    return &wrapper_memory;
}

__attribute__((noinline))
void* WASM_EXPORT(alloc)(size_t size) {
    // user defined its malloc function to use
    return malloc(size);
}

void WASM_EXPORT(dealloc)(void* ptr) {
    // user define how to free with its malloc
    return free(ptr);
}

// The smart contract
void WASM_EXPORT(droplet_entry)(void *data, size_t len, uint64_t user) {
   // To implement by user
}

// The smart contract installation
void WASM_EXPORT(droplet_install)(void *data, size_t len)
{
    // To implement by user
}

// Function to dump the content
void WASM_EXPORT(dump)(int fd) {
    //
}

// Function used in testing runtime
uint64_t WASM_EXPORT(entry_ret_u64)(void* data, size_t size) {
    //Only used in this work runtime to be able to directly have a return value
}
```

Build and Execution Instructions

The symbolic compilation runtime and all associated tooling for this work are available at:
https://github.com/2Tricky4u/SemesterProject

Environment Requirements and Dev Container

To ensure compatibility with the LLVM APIs used by `inkwell`, the runtime must be built using LLVM version 16. A ready-to-use Docker-based development container is provided for this purpose in .devcontainer.

Dockerfile. The container sets up a full Rust and LLVM 16 toolchain in Ubuntu 22.04:

Listing 5: Excerpt from Dockerfile

```
FROM ubuntu:22.04
# ... (install LLVM 16, Rust, build tools)
ENV LLVM_SYS_160_PREFIX="/usr/lib/llvm-16"
WORKDIR /project
```

devcontainer.json. The development container includes IDE integration with Rust Analyzer and LLDB:

Listing 6: Excerpt from devcontainer.json

```
{
  "name": "droplet-devcontainer",
  "image": "ubuntu:22.04",
  "features": {
    "rust": {"version": "stable"},
    ...
  },
  "postCreateCommand": "bash␣.devcontainer/setup.sh"
}
```

Feature Flags and Compilation Options

The symbolic pipeline is feature-gated to allow fine-grained control over optimization and UTX emission strategies. Relevant flags in `Cargo.toml` include:

- `test-entrypoint` – enables the contract test entry usage (*entry_ret_u64*)
- `load-store-in-bound-check` – inserts naive bounds checks
- `avoid-already-checked`, `naive-intra`, `inter-check-opt` – progressively enable intra- and inter-block memory optimizations
- `base-ptr-opt` - avoid the memory check access to base pointer if assumed as safe
- `llvm-opti` - enable the clang optimisation of code needed for current optimizations
- `inline-memory-check` - annotate llvm .bc *mem_check* function to recommend inlining instead of call
- `utxemit-v1`, `utxemit-v2`, `utxemit-v3` – control UTX emission variants

Build Pipeline Overview

Benchmarks are compiled and transformed in four phases:

1. **C to Wasm:** Each `.c` benchmark is compiled to `.wasm` with Clang using the Wasm64 backend and optimizations disabled as needed.
2. **Droplet Transform:** For each configuration (e.g., `opt1`, `opt2`, etc.), `cargo build` compiles the `droplet` binary with selected feature flags, which is then used to transform `.wasm` files into LLVM `.bc` bitcode.
3. **Link to `.so`:** The bitcode is linked with `trap.o` and compiled into shared libraries using Clang-16.
4. **Execution:** The resulting `.so` files can be executed via the testing runtime to gather benchmark traces.

Automated Build ScriptA complete pipeline script automates this process and logs outputs for reproducibility as compile_sandox.sh in test folder. Failed steps are logged in `test/sandbox/logs/`, and successful builds are emitted to `test/sandbox/obj/`.

To reproduce a manual compilation, follow the steps given below.

Listing 7: Snippet for .so compilation

```
clang-16 -O2 --target=wasm64 -D__WASM__ -c file.c -o file.wasm
cargo build --features "test-entrypoint base-ptr-opt" --package droplet
droplet file.wasm file.bc
clang-16 -O0 -shared file.bc trap.o -o file.so
```

This setup ensures reproducible experiments and easy toggling of symbolic optimization passes for comprehensive benchmark evaluation.