CIMA-R: Compiler-Enforced Resilience and Recovery Against Memory Safety Attacks in Cyber-Physical Systems

Ethan McKean
University of Michigan

Andre Quimper Osores
University of Michigan

Forest Zhang
University of Michigan

Abstract

Cyber-physical systems (CPS) frequently run memory-unsafe C/C++ code, where invalid memory accesses can jeopardize both safety and availability. Existing solutions either crash (ASan [6]), or do not land the program in a safe state after mitigating the attack (CIMA [2]).

We propose CIMA-R, a recovery-oriented extension to CIMA that improves resilience while preserving low overhead. Nearest Valid Memory recovers plausible values by searching for nearby valid memory when an invalid load occurs, and Dynamic Taint Tracking prevents tainted values from influencing non-local system state. These mechanisms address key failure cases of CIMA, including missing previous values and unsafe actuation decisions. Initial evaluation shows that CIMA-R maintains tolerable overheads while enabling safe recovery in scenarios where CIMA fails, demonstrating its practicality for real-time CPS.

All authors contributed equally to the project.

1 Introduction

Cyber-Physical systems usually have critical real time constraints that they must meet to prevent system failure. To achieve this, they are often written in memory unsafe C or C++. Bugs in production code written in C/C++ can cause memory corruption or runtime crashes.

Most mitigations for memory unsafety often rely on aborting the program, which could be catastrophic for most CPS. Therefore, alternative methods need to be developed to mitigate memory safety vulnerabilities in CPS. One such mitigation is CIMA [2]. CIMA relies on modifying the Control Flow Graph (CFG) and runtime monitoring to detect and skip loads and stores to invalid memory. Specifically stores and loads are instrumented with ASan [6], if ASan detects that a memory access is invalid during runtime, CIMA skips the respective memory instruction.

While this guarantees that the system won't crash due to an invalid memory access, it provides no guarantees that the state of the system remains sane after a load or store has been skipped. Our research, CIMA-R, proposes two alternative strategies to try to recover the state of the system after a violation is detected.

Our first strategy leverages spatial locality to minimize data divergence. In many attack models, the attacker is not able to arbitrarily introduce errors, so the unsafe action is still correlated somewhat to the real signal. Instead of simply skipping an invalid load instruction, which may introduce undefined values into the execution pipeline, CIMA-R identifies and returns the value from the nearest valid memory address to the invalid access. This approach relies on the heuristic that valid data located spatially close to the error is statistically more likely to approximate the intended value than a skipped instruction or a zero-initialization.

Our second strategy utilizes Dynamic Taint Tracking (DTT) to prevent data dependent on an invalid load from modifying global state. While preventing a crash is the primary goal, preventing the corruption of the global system state is the secondary imperative. CIMA-R tags any value resulting from a handled memory violation as "tainted." It then tracks the propagation of this taint through the program's execution. If the system detects an attempt to write a tainted value to the global state, the operation is skipped. This ensures that while the local calculation may be incorrect, the error does not propagate to critical system-wide control variables.

Finally, we evaluate the performance and safety of these approaches and compare them to the base CIMA implementation.

1.1 Related Work

Here are some other potential solutions to the problems outlined in section 1.

1.1.1 CIMA

CIMA (Compiler-Enforced Resilience Against Memory Safety Attacks) serves as the baseline architectural foundation for our work [2]. It functions by instrumenting the target application's Control Flow Graph (CFG) and utilizing runtime monitoring to intercede before memory violations occur. Specifically, CIMA leverages AddressSanitizer (ASan) metadata to check the validity of memory accesses; if a violation is detected, the system suppresses the crashing instruction rather than aborting the process. While this successfully preserves system availability, a critical requirement for Cyber-Physical Systems (CPS), it does not attempt to rectify the data flow. Consequently, CIMA leaves the system in an indeterminate state, where the omission of a critical write or the return of a default value could lead to logical divergence in the control loop.

1.1.2 Other Related Works

ASan AddressSanitizer (ASan) is a widely used dynamic testing tool designed to detect memory corruption bugs such as buffer overflows and use-after-free errors [6]. It works by shadowing the application's memory to track the state of allocated bytes, surrounding valid memory regions with "redzones" to catch out-of-bounds accesses. While highly effective for debugging during the development phase, ASan is generally ill-suited for production CPS environments in its default configuration. Its primary mitigation strategy is to immediately abort the program upon detecting a fault to prevent exploitation. For a CPS controlling physical machinery, such a fail-stop behavior can be as dangerous as the vulnerability itself, potentially leaving the physical plant in an unsafe state.

Static Program Analysis Static analysis tools, such as the Astrée analyzer, attempt to verify the absence of runtime errors (RTE) without executing the program, relying instead on abstract interpretation [3]. These tools are frequently employed in safety-critical domains, such as avionics, to mathematically prove that control software is free from specific classes of memory errors. However, static analysis is inherently limited by the trade-off between precision and scalability, often resulting in false positives or the inability to analyze complex pointer arithmetic and dynamic memory allocation found in modern C++ codebases. Furthermore, static analysis is a pre-deployment measure; it cannot protect the system against runtime hardware faults, bit-flips, or environmental edge cases that were not modeled during the analysis phase.

Self-healing Self-healing systems, particularly those utilizing failure-oblivious computing, attempt to keep a system running by manufacturing values for invalid reads and discarding invalid writes [5]. This approach shares similarities with CIMA, as the primary goal is to sustain execution through unexpected memory errors by modifying the compilation process to insert check-and-handle logic. However, traditional failure-oblivious computing often returns arbitrary pre-defined values (such as a sequence of integers) for invalid reads, which pays little regard to the semantic context of the error. In a CPS context, feeding arbitrary data into a feedback control loop can lead to "semantic crashes," where the software continues to run, but the physical actuation behaves erratically or dangerously.

Live Patching Live patching frameworks, such as Ksplice, allow administrators to apply critical security patches to a running kernel or application without requiring a reboot or downtime [1]. This is achieved by constructing a patch that pauses execution briefly to redirect functions to their new, corrected binary replacements in memory. While this is an effective maintenance strategy for minimizing downtime, it is a reactive measure rather than a proactive defense. Live patching requires that the bug has already been discovered, analyzed, and a patch released by the vendor; it offers no protection against zero-day exploits or unpatched vulnerabilities triggering during a live mission.

2 Implementation and Use Cases

The implementation details along with some potential use cases of our two main ideas are listed below.

2.1 Nearest Valid Memory (NVM)

We first propose a data approximation approach known as Nearest Valid Memory (NVM). Rather than skipping invalid memory accesses entirely or using undefined values, NVM attempts to approximate the intended value by searching for and loading from the nearest valid memory address.

The rationale is that in many cases, an invalid memory access may be spatially close to a valid region containing semantically related data. By loading from a nearby valid address, the program may continue execution with a reasonable approximation of the intended value. This approach achieves recovery through two components: compile-time instrumentation that redirects crash points to recovery logic, and a runtime search algorithm that locates the nearest accessible memory region.

2.1.1 Compile-Time Instrumentation

The NVM recovery strategy leverages the existing AddressSanitizer (ASan) infrastructure to detect invalid memory accesses at compile time. Our LLVM pass scans the program's intermediate representation for calls to `__asan_report_*` functions, which indicate locations where ASan has detected a memory safety violation that would otherwise cause a program crash.

For each detected crash point, we extract the access size (1, 2, 4, 8, or 16 bytes) from the ASan report function name. We then perform a control flow graph transformation by creating four new basic blocks:

1. **nearest_entry**: Calls the runtime helper function `__cima_find_nearest_valid` with the invalid address and access size as parameters.

2. **found_valid**: Executed when the runtime helper returns a valid address. Performs a load operation from the returned address.

3. **not_found**: Executed when no valid address is found within the search window. Uses zero as a safe fallback value.

4. **nearest_exit**: Merges the results from the *found_valid* and *not_found* paths using a PHI node.

The original branch that would lead to the ASan crash handler is redirected to the *nearest_entry* block, allowing the recovery logic to execute instead of terminating the program.

2.1.2 Runtime Search Algorithm

At runtime, the `__cima_find_nearest_valid(void* invalid_ptr, size_t access_size)` function implements a bidirectional search through the address space to locate the nearest valid memory location. The function consults ASan's shadow memory to determine validity and returns either a valid address or NULL if no valid memory exists within the search bounds.

Shadow Memory Validity AddressSanitizer maintains a shadow memory region that tracks the validity of each byte in the application's address space. For x86-64 architectures, the shadow memory mapping function is defined as:

$$\text{shadow\_addr}(a) = (a \gg 3) + \texttt{0x7FFF8000}$$

Each shadow byte s encodes the validity status of an 8-byte granule in the application's address space. We define the validity predicate $V(addr, size)$ as:

$$V(addr, size) = \begin{cases} \text{true} & \text{if } s = 0 \\ \text{true} & \text{if } 1 \leq s \leq 7 \wedge ((addr \bmod 8) + size) \leq s \\ \text{false} & \text{if } s \geq \texttt{0xF1} \end{cases}$$

where s is the shadow byte at $\text{shadow\_addr}(addr)$. A shadow value of 0 indicates that all 8 bytes of the granule are accessible. Values 1 through 7 indicate that only the first s bytes are accessible (partially valid). Values 0xF1 and higher indicate poisoned memory (e.g., freed heap regions, stack redzones).

Bidirectional Search The search algorithm attempts to find the nearest valid address by exploring addresses in order of increasing distance from the invalid access. Formally, the search objective is:

$$\text{find } a^* = \arg\min_{a \in S} |a - a_{\text{invalid}}|$$

where $S = \{a : V(a, \text{size}) \wedge |a - a_{\text{invalid}}| \leq 4096\}$ is the set of valid addresses within a 4KB window (±2048 bytes from the invalid address). This bound limits the performance overhead of the search while still covering nearby memory regions that are likely to contain related data.

The algorithm proceeds as follows:

1. Compute the base granule: $g_{\text{base}} = a_{\text{invalid}} \gg 3$
2. For offset $\delta = 0, 1, 2, \ldots, 512$:
   (a) Check forward direction: $a_f = (g_{\text{base}} + \delta) \ll 3$
   (b) If $V(a_f, \text{size})$ is true, return a_f
   (c) If $\delta > 0$, check backward direction: $a_b = (g_{\text{base}} - \delta) \ll 3$
   (d) If $V(a_b, \text{size})$ is true, return a_b
3. If no valid address is found after 512 iterations, return NULL

The bidirectional nature ensures that closer addresses are checked first, regardless of whether they are above or below the invalid address in memory. The search operates on 8-byte aligned granules to match ASan's shadow memory encoding.

Value Recovery Upon return from the runtime search function, the compile-time instrumentation determines the recovered value v as follows:

$$v = \begin{cases} \text{Load}(a^*, \text{size}) & \text{if } a^* \neq \text{NULL} \\ 0 & \text{if } a^* = \text{NULL} \end{cases}$$

This value is then merged with the value from the normal (non-crashing) execution path using a PHI node at the *nearest_exit* block, allowing subsequent program instructions to use either the valid loaded value or the recovered approximation depending on which path was taken.

2.1.3 Motivating Example

In section section 3.1.1, we implement a system mirroring an embedded system loop with an ADC, PLC, and motor controller. In the results, we see that NVM is able to preserve the pattern of input when the input under or overshoots a valid range whereas CIMA and DTT (to be discussed below) fail. This results in the CPS maintaining correct behavior.

2.2 Dynamic Taint Tracking (DTT)

As an alternative to the nearest-neighbor recovery strategy, we propose a containment-focused approach known as Dynamic Taint Tracking (DTT). While the nearest-neighbor strategy attempts to approximate valid data to maintain execution flow, it risks introducing incorrect values that could lead to logic errors. In contrast, the DTT strategy prioritizes state integrity over data approximation.

Rather than attempting to guess the correct value for an invalid load, DTT marks the resulting value as unsafe. If the program subsequently attempts to use this unsafe data to modify the global system state, CIMA-R intervenes to suppress the operation, thereby containing the error to the local scope.

2.2.1 Core Definitions

We define our Dynamic Taint Tracking (DTT) policy using a taint status function $T(x)$, where $T(x) = 1$ indicates that a register or memory location x is tainted (unsafe), and $T(x) = 0$ indicates it is clean.

Sources A source is the origin of tainted data. Let r_{dst} be the destination register of a load instruction from memory address m. If CIMA-R detects that the memory access is invalid, the instruction is skipped, and the destination register is marked as a source:

$$T(r_{dst}) = \begin{cases} 1 & \text{if load from } m \text{ is invalid} \\ 0 & \text{otherwise} \end{cases}$$

Taint Propagation Once taint enters the system, it propagates via three mechanisms:

1. **Data Flow:** For any instruction, the destination register r_{dst} becomes tainted if any of its input operands are tainted.

$$T(r_{dst}) = T(operand_1) \vee T(operand_2) \vee \dots$$

2. **Control Dependence:** If the evaluation of a branch condition depends on a tainted register r_{cond}, then any register r_{dep} defined within that branch is considered tainted.

$$T(r_{cond}) = 1 \implies T(r_{dep}) = 1$$

3. **Stack Memory:** Local variables on the stack are tracked to allow taint to persist through spills. If a tainted register r_{src} is stored to a stack address m_{stack}, that memory location becomes tainted.

$$T(m_{stack}) \leftarrow T(r_{src})$$

Sinks A sink is a critical operation that must be protected. We define a sink as any store operation to **non-stack** (global or heap) memory. If an instruction attempts to store a value from register r_{val} to a global memory address m_{global}, the operation is skipped if the value is tainted.

$$\text{Action}(store \to m_{global}) = \begin{cases} \text{Skip} & \text{if } T(r_{val}) = 1 \\ \text{Execute} & \text{if } T(r_{val}) = 0 \end{cases}$$

2.2.2 Motivating Example

As shown in section section 3.1.2, there are cases where trying to approximate a memory value or use the last definition of a variable results in undesirable behavior. In section 3.1.2 we implement a Proportional Control System, and demonstrate that DTT protects the physical state of the program, whereas Nearest Neighbor and base CIMA both fail. This is because using a close valid memory address or the last defined value of a Proportional Controller can cause large over-corrections, whereas DTT would not update the system state until we can get a reliable sensor reading.

3 Evaluation

We evaluate our implementations for both safety and latency. Our implementation and test cases can be found here[4]. We created basic correctness tests that all passed, but only discuss the case studies in this section.

3.1 Safety

We evaluate the correctness of our implementations on both case studies mentioned in section 2.1.3 and section 2.2.2.

3.1.1 Motor Controller

The Motor Controller case study functions as follows:

1. **Setup control table and fan state.** Allocate an integer array `control_array[0..9]`. Initialize entries 0–4 to low PWM values $\{10, 15, 20, 25, 30\}$ and entries 5–9 to high PWM values $\{70, 75, 80, 85, 90\}$. Initialize the fan state: current speed $S \leftarrow 0$, last PWM $\leftarrow 0$, and consecutive-zero counter $Z \leftarrow 0$.

2. **Define the ADC index generator.** For each cycle (iteration) i, compute whether it is a "high" cycle (odd i) or "low" cycle (even i), and return an ADC index:

   (i) In **normal** mode: return index 2 on low cycles and 9 on high cycles.

   (ii) In **attack** mode: return index 4 on low cycles and 11 on high cycles.

(Note: index 11 is out of bounds for `control_array[0..9]`.)

3. **Define the fan "plant" update given a PWM command.** Given a PWM value `pwm_value`, map it to a binary actuation decision:
   - (i) If PWM ≤ 40, set logic $u \leftarrow 0$ (off).
   - (ii) If PWM ≥ 60, set logic $u \leftarrow 1$ (on).
   - (iii) Otherwise (40–60), also set $u \leftarrow 0$.

   Then update the fan speed state:
   - (i) If $u = 1$: increase speed $S \leftarrow \min(S + 15, 100)$ and reset $Z \leftarrow 0$.
   - (ii) If $u = 0$: increment $Z \leftarrow Z + 1$; if $Z > 2$, decrease speed $S \leftarrow \max(S - 8, 0)$.

   Finally print whether the fan is considered RUNNING (speed $> 5\%$) or STOPPED.

4. **Phase 1: Normal ADC operation (20 cycles).** For cycles $i = 0..19$:
   - (i) Read ADC index $a \leftarrow$ `adc_read`$(i,$`ADC_NORMAL`$)$, which alternates between 2 and 9.
   - (ii) Lookup PWM command $p \leftarrow$ `control_array`$[a]$ (always in-bounds), yielding either a low PWM (index $2 \rightarrow 20$) or a high PWM (index $9 \rightarrow 90$).
   - (iii) Apply the PWM command to the fan model via `fan_speed`, updating the speed state and printing telemetry.

5. **Phase 2: ADC "attack" mode (20 cycles).** For cycles $i = 0..19$:
   - (i) Read ADC index $a \leftarrow$ `adc_read`$(i,$`ADC_ATTACK`$)$, which alternates between 4 (in-bounds) and 11 (out-of-bounds).
   - (ii) Lookup PWM command $p \leftarrow$ `control_array`$[a]$. When $a = 11$, this performs an out-of-bounds read and may yield an arbitrary PWM value.
   - (iii) Apply the (possibly corrupted) PWM command to the fan model via `fan_speed`. Depending on the read value, the controller may incorrectly command ON/OFF, causing unsafe or unintended acceleration/deceleration behavior while the program continues executing.

6. **Cleanup.** Free `control_array` and terminate.

As expected, the output for CIMA and our NVM on cycle 13 and onward look like:

```
Cycle 13: ADC=11, Control=30
  Fan Speed: PWM=30 -> Logic=0 | Speed=4.0% | STOPPED
  ...
```

V.S.

```
Cycle 13: ADC=11, Control=85
  Fan Speed: PWM=85 -> Logic=1 | Speed=100.0% | RUNNING
  ...
```

So our implementation protected against the attack while CIMA failed.

3.1.2 Water Treatment Level

The water treatment case study functions as follows:

1. **Setup.** Initialize the tank water level $L \leftarrow 0$. Set $N\_STEPS = 20$ and $\Delta t = 1.0$. Fix the drain rate to $d = 1$, the target level to $L^\star = 5.0$, and the maximum safe level to $L_{\max} = 6.0$. Define `fill_table` $= \{0.0, 0.9, 1.8, 2.7, 3.6, 4.5\}$. At timesteps $t \in \{7, 8, 9\}$, inject an attack that increases the controller index by 6, causing an out-of-bounds table lookup.

2. **For each discrete time step $t = 0, 1, \ldots, N\_STEPS - 1$, do:**
   - (i) **Measure error (sensor).** Compute the level error $e \leftarrow L^\star - L$.
   - (ii) **Quantize error to an index.** Set $i \leftarrow \lfloor e \rfloor$ (implemented by casting `(int)e`).
   - (iii) **Clamp negative indices.** If $i < 0$, set $i \leftarrow 0$.
   - (iv) **Attack injection (index corruption).** If $t \in$ [`ATTACK_STEP`, `ATTACK_STEP` $+ 2$], set $i \leftarrow i + 6$.
   - (v) **Compute control action (lookup-table controller).** Read the inflow command $u \leftarrow$ `fill_table`$[i]$. (No upper-bound check is performed, so this is an out-of-bounds read when $i \geq 6$.)
   - (vi) **Plant update (tank dynamics).** Update the tank level using
   $$L \leftarrow L - d + u \cdot \Delta t.$$
   - (vii) **Enforce nonnegativity.** If $L < 0$, set $L \leftarrow 0$.
   - (viii) **Report and safety check.** Output (t, e, i, u, L), and flag a safety violation if $L > L_{\max}$.

As expected, the output for CIMA and our DTT on iterations 7-9 is:

```
t= 7  sensor= 1.20  idx= 7  fill=1.80  level= 4.60
t= 8  sensor= 0.40  idx= 6  fill=1.80  level= 5.40
t= 9  sensor=-0.40  idx= 6  fill=1.80  level= 6.20
*** ABOVE MAX_LEVEL (6.00) ***
```

V.S.

```
t= 7  sensor= 1.20  idx= 7  fill=1.80  level= 2.80
t= 8  sensor= 2.20  idx= 8  fill=1.80  level= 1.80
t= 9  sensor= 3.20  idx= 9  fill=1.80  level= 0.80
```

So our implementation protected against the attack while CIMA failed.

3.2 Latency

Beyond theoretically fixing the memory safety issues, our solutions also must be within the tolerable cycle time of the CPS. This section analysis the latency overhead of our solutions to see if they would be feasible in a real-world CPS. All benchmarks were run on an x86 64-bit machine with an Intel(R) Xeon(R) CPU E3-1275 v6 @ 3.80GHz and 62 Gigabytes of RAM.

3.2.1 Our Case Studies

We provide benchmarks for each case study run with no CIMA, base CIMA, nearest valid, and taint tracking. Below, we track the maximum time taken to run the tests over 100 trials. In order to judge the latency over normal runs and attack, we track both in our water treatment tests.

From fig. 1, wee see that our solutions impose very little overhead for the most part on top of base CIMA. The exception is DTT on the ADC example. The dominant source of overhead for DTT is the allocation and access of excessively large shadow memory regions. Our taint tracking mirrors the original data layout rather than using a compact representation, which significantly increases the effective working set size. On memory-bandwidth-bound workloads, this additional memory traffic overwhelms the baseline computation and amplifies cache and DRAM pressure, leading to substantial slowdowns. This effect is less pronounced in compute-heavy or control-dominated workloads, where memory footprint is not the primary bottleneck. This effect can be mitigated by controlling the size of shadow memory regions or integrating our solution into ASan.

3.2.2 Stream Benchmarks

STREAM benchmarks measure sustained main-memory bandwidth by executing simple, streaming vector kernels over large arrays that exceed cache capacity, thereby isolating memory subsystem performance. In our evaluation, we use STREAM to assess the worst-case memory-bandwidth impact of our instrumentation by reporting the slowest observed execution across runs rather than average throughput. Focusing on worst-case behavior provides a conservative estimate of overhead, which is more appropriate for safety-critical and real-time systems where performance degradation in adverse conditions is more relevant than typical-case behavior.

The STREAM benchmark executes four simple memory-bound kernels (Copy, Scale, Add, and Triad), each run 10 times over large arrays that exceed cache capacity with the first iteration treated as a warm-up and excluded from the results. STREAM measures sustained memory bandwidth by timing full-array streaming loads and stores. We display the worse time recorded for our safe-latency purposes.

In fig. 2, we see again that NVM incures a negligibe amount of overhead compared to base CIMA. However, DTT again sees a lot of latency because its large shadow memory footprint dominates performance on memory-bandwidth-limited workloads.

3.2.3 Comparison with Base CIMA

Our implementation of base CIMA adds slightly more latency than what the original paper observed. This may be due to our lack of actual physical processes. Still, the additional latency is comparable. In the original CIMA paper, they found that for their secure water treatment plant (SWaT) and secure urban traffic system (SecUTS) use cases, the CIMA execution consumed at most 3.2 ms of a 10 ms cycle time and at most 2.5 ms of a 30 ms cycle time, respectively.

Scaling our additional solutions off of our implementation of CIMA and comparing it to the real CPS's used in the original paper, we see an approximately $1.23\times$ increase for NVM compared to base CIMA in the worse case, which translates

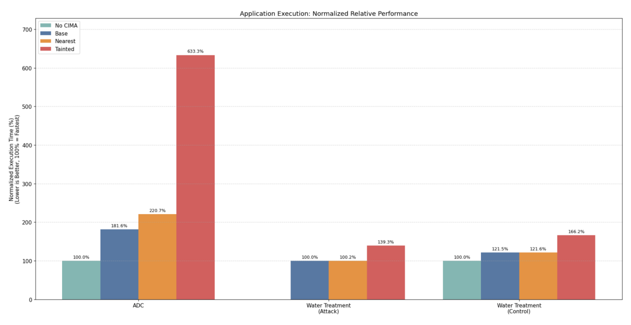

Figure 1: Case study latency normalized to control (No CIMA).

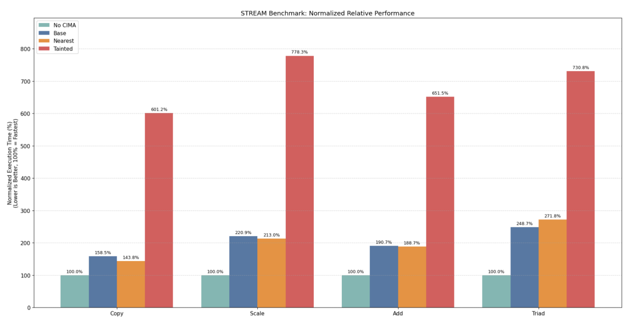

Figure 2: STREAM benchmarks normalized to control (No CIMA).

to a raw cycle time of 3.94 ms and 3.08 ms respectively in the real CPS's. For DTT, in the water treatment example, we see an approximately 1.37× increase compared to base CIMA, which translates to a raw cycle time of 4.38 ms and 3.43 ms in the real CPS's. All of these are under the max cycle time needed.

However, for more memory intensive programs, like the ADC test or STREAM benchmarks, we see a nearly 3.5× increase compared to base CIMA, which is above the threshold for SWaT, but is tolerable for SecUTS. Most CPS systems are not this memory intensive, so the solution should still work for most use cases.

4 Conclusion

In this work, we show that availability-oriented defenses such as CIMA are not sufficient to ensure safe behavior in cyber-physical systems. We identify concrete scenarios in which CIMA fails to preserve meaningful system behavior, while our proposed recovery mechanisms succeed. In particular, we demonstrate that when the cost of stalling is low, when malformed sensor signals remain correlated with true values, or when no reliable previous value exists, simple instruction skipping can lead to unsafe or semantically incorrect actuation decisions.

To address these limitations, we introduce recovery-oriented extensions that either approximate plausible values or prevent corrupted data from influencing global system state. Our evaluation on representative CPS case studies shows that these approaches can maintain safe physical behavior in cases where CIMA alone fails, while allowing the system to continue operating.

Finally, we show that our techniques incur tolerable overheads for less memory-intensive applications, making them practical for a broad class of real-time CPS workloads. Together, these results suggest that recovery-aware compiler and runtime mechanisms are a necessary complement to fail-stop or failure-oblivious approaches, and that preserving physical safety requires reasoning beyond mere crash avoidance.

5 Future Work

There are several promising directions for extending this work.

Deciding which solution to use. One natural avenue is to enrich the compiler analysis to automatically select the most appropriate mitigation strategy for a given code region. Neither CIMA nor our proposed recovery mechanisms are universally optimal: nearest-valid recovery may preserve progress in some contexts, while dynamic taint tracking is more appropriate in others. A more expressive compiler could analyze control structure, data dependencies, and actuation paths to decide which strategy best preserves safety. In particular, recent advances in large language models suggest the possibility of incorporating semantic reasoning about code structure and intent to guide this selection, enabling more adaptive and context-aware resilience mechanisms.

Extending DTT. A second direction is to extend taint tracking beyond function boundaries. Our current design focuses on preventing tainted values from corrupting global state within well-defined abstractions, but taint may propagate across function calls, modules, or asynchronous control paths. Tracking taint interprocedurally and across global memory would provide stronger guarantees that memory-safety violations cannot influence system-wide state, even when control logic spans multiple components.

Another natural direction is to improve the efficiency of taint tracking by integrating shadow memory management directly into AddressSanitizer's existing shadow infrastructure. ASan already maintains a compact, byte-granular shadow mapping that encodes the validity of application memory with low overhead. By reusing or extending this shadow representation to encode taint information, rather than allocating separate shadow regions that mirror the original data layout, we could significantly reduce memory footprint and bandwidth overhead. Such an integration would allow taint metadata to be stored and accessed in a cache-friendly manner, while leveraging ASan's mature address translation and instrumentation logic, potentially bringing the steady-state overhead of taint tracking closer to that of CIMA on memory-intensive workloads.

Correctness Models. Finally, our evaluation highlights the need for more rigorous correctness models for cyber-physical systems. Current defenses are largely framed in terms of availability (avoiding crashes), yet continued execution alone does not guarantee safe physical behavior. Developing formal models of acceptable physical-state evolution, rather than ad hoc safety thresholds, would provide a principled foundation for future compiler and runtime designs. Such models could guide the construction of recovery policies that are provably aligned with system-level safety objectives, rather than merely preserving execution.

References

[1] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 187–198, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584829. doi: 10.1145/1519065.1519085. URL https://doi-org.proxy.lib.umich.edu/10.1145/1519065.1519085.

[2] E. G. Chekole, S. Chattopadhyay, M. Ochoa, H. Guo, and U. Cheramangalath. Cima: Compiler-enforced resilience against memory safety attacks in cyber-physical systems. *Computers & Security*, 94:101832, 2020. doi: 10.1016/j.cose.2020.101832. https://doi.org/10.1016/j.cose.2020.101832.

[3] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In M. Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0.

[4] E. McKean, A. Q. Osores, and F. Zhang. ethanmckean/cimar-llvm, Dec. 2025. URL https://github.com/ethanmckean/cimar-llvm.

[5] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through Failure-Oblivious computing. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, Dec. 2004. USENIX Association. URL https://www.usenix.org/conference/osdi-04/enhancing-server-availability-and-security-through-failure-oblivious-computing.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 309–318. USENIX Association, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.