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

Eyasu Getahun Chekole[a,b], Sudipta Chattopadhyay[b], Martín Ochoa[b,c], Huaqun Guo[a], Unnikrishnan Cheramangalath[b]

[a]*Institute for Infocomm Research, Singapore 138632, Singapore*
[b]*Singapore University of Technology and Design 487372, Singapore*
[c]*AppGate Inc, Bogotá, Colombia*

Abstract

Memory-safety attacks have been one of the most critical threats against computing systems. Although a wide-range of defense techniques have been developed against these attacks, the existing mitigation strategies have several limitations. In particular, most of the existing mitigation approaches are based on aborting or restarting the victim program when a memory-safety attack is detected, thus making the system unavailable. This might not be acceptable in systems with stringent timing constraints, such as cyber-physical systems (CPS), since the system unavailability leaves the control system in an unsafe state. To address this problem, we propose CIMA – a resilient mitigation technique that prevents invalid memory accesses at runtime. CIMA manipulates the compiler-generated control-flow graph to automatically detect and bypass unsafe memory accesses at runtime, thereby mitigating memory-safety attacks along the process. An appealing feature of CIMA is that it significantly improves system availability and resilience of the CPS even under the presence of memory-safety attacks. To this end, we design our experimental setup based on a realistic Secure Water Treatment (SWaT) and Secure Urban Transportation System (SecUTS) testbeds and evaluate the effectiveness and the efficiency of our approach. The experimental results reveal that CIMA handles memory-safety attacks effectively while meeting the real-time constraints and physical-state resiliency of the CPS under test. Using CIMA, we have also discovered a memory-safety vulnerability in the firmware of programmable logic controllers and a CVE ID has already been assigned for it.

Keywords:
Cyber-Physical Systems Security, CPS Security, Memory-Safety Attacks, Software Security, Critical Infrastructures Security, Mitigation, Efficiency, Resilience, Availability

1. Introduction

Software systems with stringent real-time constraints are often written in C/C++ since they aid in generating efficient program binaries. However, since memory management is handled manually in C/C++, programs written in such languages often suffer from memory-safety vulnerabilities, such as buffer over/underflows, dangling pointers, double-free errors and memory leaks. Due to the difficulty in discovering these vulnerabilities, they might often slip into production run. This leads to memory corruptions and runtime crashes even in deployed systems. In the worst case, memory-safety vulnerabilities can be exploited by a class of cyber attacks, which we refer to as *memory-safety attacks* [1, 2]. Memory-safety attacks, such as code injection [3, 4] and code reuse [5, 6, 7], can cause devastating effects by compromising vulnerable programs in any computing system. These attacks can hijack or subvert specific operations of a system or take over the entire system, in general. In certain domains, such as in Cyber-Physical Systems (CPS), memory-safety attacks can cause significant physical impact to the CPS by subverting the control logic. A detailed account of such attacks is provided in existing works [7, 8].

Given a system vulnerable to memory-safety attacks, its runtime behavior will depend on the type of memory accesses being made, among others. For instance, accesses to an array element beyond the imposed length of the array (buffer overflows) could be exploited to overwrite the return address of a function. However, a safety check generated at compile time [9] can be added before any memory access to ensure validity of the memory going to be accessed. This can be accomplished via compiler-assisted program analysis. Traditionally, such memory-safety compilers will generate an exception and abort the program when such violations are found at run-time. In the past, this approach has been proposed for improving the security of CPS [8, 10] and it has been shown that it can be useful to prevent exploitation in the traditional sense (i.e. injecting arbitrary malicious logic to a PLC). However, such an approach, while preventing certain types of exploitation, can be in turn exploited for *availability attacks*, which aim to make the victim system unavailable for an arbitrary duration. In CPS, availability is one of the most importante security properties to preserve, often more than confidentiality [11], since system unavailability leads the control system to an unsafe state and, in return, may cause system disruption and physical damages. Traditional availability attacks such as Denial-of-Service (DoS) attacks can be detected by an intrusion detection system (IDS). However, it is more difficult to detect an availability attack triggered by a memory-safety vulnerability, since one carefully crafted packet

would suffice to crash the programmable logic controller (PLC) in CPS.

In this paper, we propose a strategy to offer more resilience to availability attacks that are triggered by exploiting memory safety vulnerabilities of CPS components such as PLCs. Concretely, we avoid aborting the vulnerable program when detecting a memory safety violation at runtime. Instead, our proposed approach proactively prevents memory-safety violations by *bypassing* the illegal instructions, *i.e., instructions that attempt to access memory illegally*. In such a fashion, our approach favors the availability of the targeted system. We argue that, although by doing so we are technically modifying the low-level semantics of the system under attack, we do so only for combinations of inputs and states that lead to a memory-safety violation, and that are thus outside the "intended" program semantics, which is undefined in those cases.

Although our approach is conceptually simple, it is however technically challenging to bypass the manifestation of illegal memory access. This is because, such a strategy demands full control to manipulate the normal flow of program execution. To bypass the execution of certain instructions, we need modifications to the control flow of the program. From a technical perspective, such modifications involve the manipulation of program control flow over multitudes of passes in mainstream compilers.

To alleviate the technical challenges, CIMA systematically combines compile-time instrumentation and runtime monitoring to defend against memory-safety attacks. Specifically, at compile time, each instrumented memory access is guarded via a conditional check to detect its validity at runtime. In the event where the conditional check fails, CIMA skips the respective memory access at runtime. To implement such a twisted flow of control, CIMA automatically transforms the program control flow logic within mainstream compilers. This makes CIMA a proactive mitigation strategy against a large class of memory-safety attacks.

It is the novel mitigation strategy and the evaluatuon on realistic CPS testbeds that sets our CIMA approach apart from existing related works [9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Most of the existing mitigation schemes against memory-safety attacks are primarily programmed to abort or restart the victim system when an attack or a memory-safety violation is detected. Other schemes, such as self-healing [25, 26, 27] or live patching [28], are based on directly detecting exploitations or attacks and resuming the corrupted system from a previous safe state. CIMA never aborts the system and avoids the heavy overhead of maintaining system states for checkpoints. On the contrary, CIMA follows a fundamentally different approach, i.e., to continue execution by skipping *only* the illegal instructions. In general, CIMA is a resilient mitigation strategy that effectively prevents memory-safety attacks and improves system availability with acceptable overhead.

Although our proposed security solution is applicable to any computing system that involves C/C++ programs, we mainly focus on the CPS domain. This is because, unlike mainstream systems, CPS often imposes conflicting design constraints involving real-time guarantees and physical-state resiliency involving its physical dynamics and security. In this work, we also formally model these design constraints and use them as benchmarks to evaluate the efficiency and resilience of our proposed solution.

In CPS, the memory-safety vulnerabilities might be found in the firmware (or sometimes in the control software) of PLCs. Such firmware is commonly implemented in C/C++ for the sake of efficiency. Consequently, it is not uncommon to have buffer over/underflows and dangling pointers being regularly discovered even in modern PLCs. In fact, recent trends in Common Vulnerabilities and Exposures (CVEs) show the high volume of interest in exploiting these vulnerabilities in PLCs [29, 30, 31, 32, 33, 34, 35, 36]. This shows that the mitigation of memory-safety attacks in CPS should not merely be restricted to academic research. Instead, it is a domain that requires urgent and practical security solutions to protect a variety of critical infrastructures at hand. Nonetheless, attacks that manipulate sensor and actuator values in CPS (either directly on sensors/actuators or on communication channels) are orthogonal issues that are out of our scope, and that can be mitigated using for instance physical invariants [37] or model-based approaches [38, 39, 40].

In summary, *our work tackles the problem of ensuring critical systems and services to remain available and effective while successfully mitigating a wide-range of memory-safety attacks*. We offer a concrete implementation of our approach and an evaluation in SWaT and SecUTS testbeds. We also discuss limitations of our approach. We make the following technical contributions:

1. We effectively prevent system crashes that could be arisen due to memory-safety violations.
2. We effectively and efficiently prevent memory-safety attacks in any computing system.
3. We define the notion of physical-state resiliency that is crucial for CPS and should be met alongside strong security guarantees.
4. Our mitigation solution ensures physical-state resiliency and system availability with reasonable runtime and storage overheads. Thus, it is practically applicable to systems with stringent timing constraints, such as CPS. To the best of our knowledge, such a dual combination of achieving memory-safety and meeting real-time and availability constraints of the CPS is a feat not accomplished by any existing work.
5. We evaluate the effectiveness and efficiency of our approach on two realistic CPS testbeds.

2. Background

In this section, we introduce the relevant background in the context of our CIMA approach.

2.1. CPS

CPS constitutes of complex interactions between entities in the physical space and the cyber space over communication

networks. Unlike traditional IT systems, such complex interactions are accomplished via communication with the physical world via sensors and with the digital world via controllers (PLCs) and other embedded devices. CPS usually impose hard real-time constraints. If such real-time constraints are not met, then the underlying system might run into an unstable and unsafe state. Moreover, the devices in a typical CPS are also resource constrained. For example, PLCs and I/O devices have limited memory and computational power. In general, a typical CPS consists of the following entities:

- **Physical plants**: Physical systems where the actual processes take place.
- **Sensors**: Devices that are capable of reading or observing information from plants or physical processes.
- **PLCs**: Controller devices that receive sensor inputs, make decisions and issue control commands to actuators.
- **Actuators**: Physical entities that are capable of implementing the control commands issued by the PLCs.
- **Communication networks**: The communication medium through which packets containing sensor inputs, control commands, diagnostic information and alarms transmit from one CPS entity to another.
- **SCADA**: A software entity designed for monitoring and controlling different processes in a CPS. It often comprises a human-machine interface (HMI) and a historian server. The HMI is used to display state information of plants and physical processes in the CPS. The historian server is used to store all operational data and the history of alarms.

An abstraction of a typical CPS architecture is shown in Figure 1. In Figure 1, x denotes the physical state of the plant, y captures the sensor measurements and u denotes the control command computed by the PLC at any given point of time.

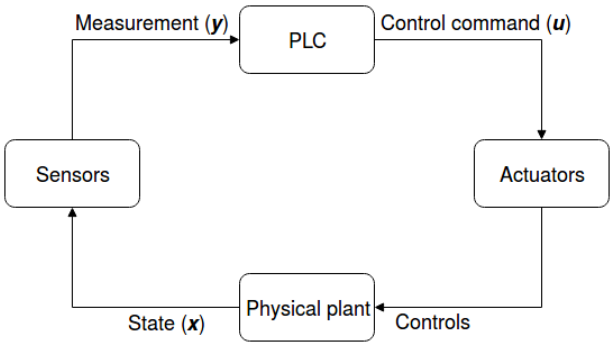


Figure 1: An abstraction of a CPS model

2.2. *Memory-safety tools*

A wide-range of memory-safety tools have been developed against memory-safety attacks. However, their applicability in CPS is limited due to several reasons. This includes the lack of error coverage, significant performance overhead, ineffective mitigation strategy and other technical incompatibility issues. After researching and experimenting on various memory-safety tools, we chose ASan [9] as our memory error detector tool. Our choice is motivated by its broad error coverage, high detection accuracy and relatively low runtime overhead when compared with other code-instrumentation based tools [9, 41].

ASan is a compile-time memory-safety tool based on code instrumentation. It instruments C/C++ programs at compile time and creates poisoned (i.e. illegal) memory regions, known as *redzones*, between any two stack, heap and global variables. As these redzones are not addressable by the running program, any attempt to access them is an illegal behavior and will be detected as a memory-safety violation. In this way, the ASan instrumented program will contain additional ASan libraries, which are checked to detect possible memory-safety violation at runtime. Such an instrumented code can detect buffer over/underflows, use-after-free errors (dangling pointers), use-after-return errors, initialization order bugs and memory leaks.

Since ASan was primarily designed for x86 architectures, it has compatibility issues with RISC-based ARM or AVR based architectures. Therefore, we adapted ASan for ARM-based architecture in our system. ASan may also rarely miss some memory violations if an attacker manages to jump over certain redzones and corrupt regions outside the redzones [9], but with very low probability. However, such issues are already addressed by pointer-based tools, e.g., SoftBoundCETS [18, 19], which we intend to use in our future experimentation.

The other major limitation of ASan is its ineffective mitigation strategy; it simply aborts the system whenever a memory-safety violation or an attack is detected. This makes ASan inapplicable in systems with stringent availability constraints, such as CPS. Due to this reason, ASan is often considered as rather a debugging tool than a runtime monitoring tool. However, using ASan only as a debugging tool does not guarantee memory-safety. This is because most memory-safety vulnerabilities (e.g. buffer overflows) can be probed by systematically crafted inputs by attackers at runtime. Since carefully tailored inputs might not be used during debugging, ASan might miss important memory bugs, which can be exploited by an attacker at runtime. Therefore, only debugging the program does not offer sufficient guarantee against memory-safety attacks or violations.

In our CIMA approach, we adopt ASan for the dual purpose of debugging and runtime monitoring, with the specific focus on mitigating memory-safety vulnerabilities. As a runtime monitoring tool, CIMA leverages on ASan to detect attacker injected memory-safety violations. Moreover, CIMA enhances the capability of ASan to mitigate memory-safety bugs on-the-fly. This is to ensure the availability of the underlying system and in stark contrast to plain system abort.

3. Attacker and system models

In this section, we first discuss the attacker model. Then, we discuss the different traits of formally modeling our system and the related design constraints.

3.1. Attacker model

The main objective of memory-safety attacks, such as code injection and code reuse attacks, is to get privileged access or take control (otherwise to hijack/subvert specific operations) of the vulnerable system. To achieve this, the attacker exploits memory-safety vulnerabilities, such as buffer overflows and dangling pointers that can be found in the targeted program. To illustrate the exploitation strategies with an example, we provide a simple C/C++ code snippet shown in Program 1. A high-level memory layout of the program is demonstrated in Figure 2a, displaying only a few memory addresses that are relevant to illustrate the attack. This includes the defined buffer and extended instruction pointer (EIP), which is the return address of the program.

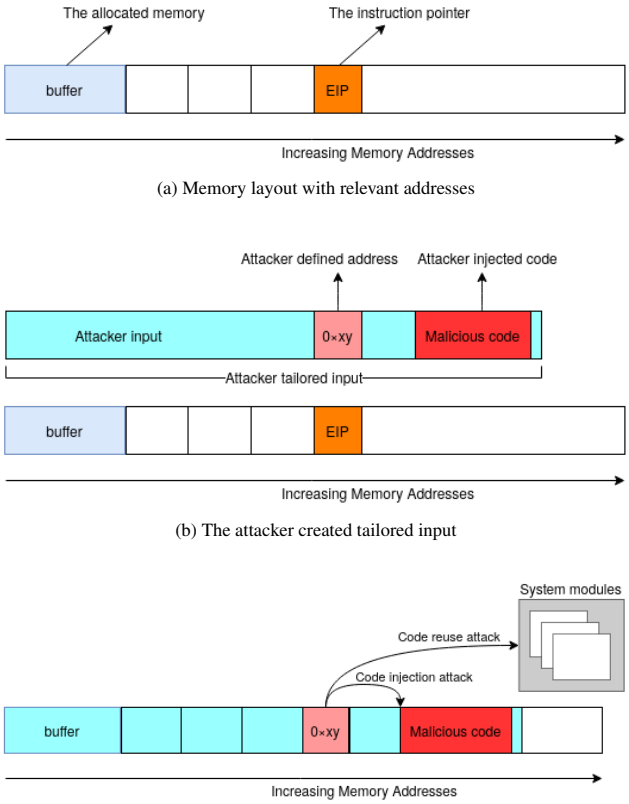
This program contains a buffer overflow vulnerability as the *"gets(buffer)"* function (in Program 1, Line 4) allows the user to provide an input that is larger than the defined buffer size. To exploit this vulnerability, the attacker creates a systematically tailored input that will serve to overwrite the buffer, the EIP and other memory addresses. Specifically, the tailored input comprises the *attacker defined address* (i.e. 0×xy in Figure 2b), that will serve to overwrite the EIP, and a *malicious code*, that will be injected in the program's address space (in case of code injection attacks). The tailored input is illustrated in Figure 2b. As demonstrated in Figure 2c, the attacker defined address (that overwrites the EIP) is made to point the start of the injected malicious code (in case of code injection attacks) or existing system modules (in case of code reuse attacks). The attacker will then send this carefully crafted input to the buffer to launch the actual attack. A detailed account of such memory-safety exploits can be found in [7, 42].

Program 1: A sample buffer overflow vulnerability

```
1 foo() {
2     char buffer[16];
3     printf("Insert input: ");
4     gets(buffer);
5 }
```

In brief, a typical memory-safety attack targeting PLCs in CPS follows the following steps (See Figure 3):

1. Interacting with the victim PLC, e.g., via network connection (for remote attacks).
2. Finding a memory-safety vulnerability (e.g., buffer overflow, dangling pointers) in the PLC firmware or control software with the objective of exploiting it.
3. Triggering a memory-safety violation on the PLC, e.g., overflowing a buffer.
4. Overwriting critical addresses of the vulnerable program, e.g., overwriting the EIP (i.e. the return address of the PLC program).
5. Using the modified address, divert control flow of the program to an injected (malicious) code (i.e. code injection attacks) or to an existing module of the vulnerable program (i.e. code reuse attacks). In the former case, the attacker may take over the PLC with the injected malicious code. In the latter case, the attacker still needs to collect appropriate gadgets from the program (basically by scanning the program's text segment), then she will synthesize a shellcode that will allow her to take over the PLC.



(a) Memory layout with relevant addresses

(b) The attacker created tailored input

(c) Diverting control to the injected code (code injection attacks) or system modules (code reuse attacks)

Figure 2: A high-level illustration of memory-safety attacks exploitation strategies

3.2. Modeling CPS design constraints

Most cyber-physical systems are highly time-critical. The communication between its different components, such as sensors, controllers (PLCs) and actuators, is synchronized by system time. Therefore, delay in these CPS nodes could result in disruption of the control system or damage the physical plant. In particular, PLCs form the main control devices and computing nodes of a typical CPS. As such, PLCs often impose hard real-time constraints to maintain the stability of the control system in a CPS.

In the following sections, we define and discuss the notions of *real-time constraints* and *physical-state resiliency*, which are crucial in the design of CPS and will be used as benchmarks to evaluate the efficiency and resilience of our mitigation strategy, respectively.

3.2.1. *Modeling real-time constraints*

In general, PLCs undergo a cyclic process called *scan cycle*. This involves three major operations: input scan, PLC program (logic) execution and output update. The time it takes to

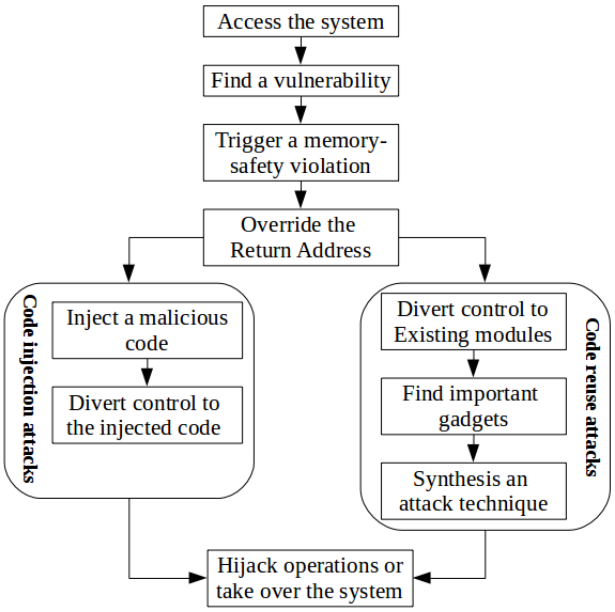


Figure 3: Overview of memory-safety attack exploitations

complete these operations is called *scan time* (T_s). A detailed account of modeling T_s can be found in [43]. A typical CPS defines and sets an upper-bound on time taken by the PLC scan cycle, called *cycle time* (T_c). This means the scan cycle must be completed within the duration of the cycle time specified, i.e., $T_s \leq T_c$. We refer this constraint as a *real-time constraint* of the PLC. By design, PLCs meet this constraint. However, due to additional overheads, such as memory-safety overheads (MSO), PLCs might not satisfy its real-time constraint. As discussed above, by hardening the PLC with our memory-safety protection (detection + mitigation), the scan time increases. This increase in the scan time is attributed to the MSO. Concretely, the memory-safety overhead is computed as follows:

$$MSO = \hat{T}_s - T_s, \tag{1}$$

where $\hat{T}_s$ and T_s are scan time with and without memory-safe compilation, respectively.

It is crucial to check whether the induced MSO by the memory-safe compilation still satisfies the real-time constraint imposed by the PLC. To this end, we compute MSO for – 1) average-case and 2) worst-case scenarios. In particular, after enabling memory-safe compilation, we compute the scan time (i.e. $\hat{T}_s$) for n different measurements and compute the respective average (i.e. $mean(\hat{T}_s)$) and worst-case (i.e. $max(\hat{T}_s)$) scan time. Formally, we say that the MSO is acceptable in average-case if the following condition is satisfied:

$$\frac{\sum_{i=1}^{n} \hat{T}_s(i)}{n} \leq T_c \tag{2}$$

where $\hat{T}_s(i)$ captures the scan time for the i-th measurement after the memory-safe compilation.

In a similar fashion, MSO is acceptable in the worst-case with the following condition:

$$\max_{i=1}^{n} \hat{T}_s(i) \leq T_c \tag{3}$$

3.2.2. *Physical-state resiliency*

The stability of CPS controllers (i.e. PLCs in our case) plays a crucial role in enforcing the dynamics of a cyber-physical system to be compliant with its requirement. For example, assume that a PLC issues control commands every T_c cycle and an actuator receives these commands at the same rate. Therefore, cycle time of the PLC is T_c. If the PLC is down for an arbitrary amount of time say τ, then the actuator will not receive fresh control commands for the duration τ. Consequently, the physical dynamics of the respective CPS will be affected for a total of $\frac{\tau}{T_c}$ scan cycles.

We note that the duration τ might be arbitrarily large. Existing solutions [9], albeit in a non-CPS environment, typically revert to aborting the underlying process or restart the entire system when a memory-safety attack is detected. Due to the critical importance of availability constraints in CPS, our CIMA approach never aborts the underlying system. Nevertheless, CIMA induces an overhead to the scan time of the PLC, as discussed in the preceding section. Consequently, the scan time of PLCs, with CIMA-enabled memory-safety (i.e. $\hat{T}_s$), may increase beyond the cycle time (i.e. T_c). In general, to accurately formulate τ (i.e. the amount of downtime for a PLC), we need to consider the following three mutually exclusive scenarios:

1. The system is aborted or restarted.
2. The system is neither aborted nor restarted and $\hat{T}_s \leq T_c$. In this case, there will be no observable impact on the physical dynamics of the system. This is because the PLCs, despite having increased scan time, still meet the real-time constraint T_c. Thus, they are not susceptible to downtime.
3. The system is neither aborted nor restarted and $\hat{T}_s > T_c$. In this scenario, the PLCs will have a downtime of $\hat{T}_s - T_c$, as the scan time violates the real-time constraint T_c.

Based on the intuitions mentioned in the preceding paragraphs, we now formally define τ, i.e., the downtime of a PLC as follows:

$$\tau = \begin{cases} \Delta, & \text{system is aborted/restarted} \\ 0, & \hat{T}_s \leq T_c \\ \hat{T}_s - T_c, & \hat{T}_s > T_c \end{cases} \tag{4}$$

where Δ captures a non-deterministic threshold on the downtime of PLCs when the underlying system is aborted or restarted.

Example. As an example, let us consider the first process in SWaT (discussed in Section 5.1). This process controls the inflow of water from an external water supply to a raw water tank. PLC1 controls this process by opening (with "ON" command) and closing (with "OFF" command) a motorized valve, i.e., the actuator, connected with the inlet pipe to the tank. If the valve is "ON" for an arbitrarily long duration, then the raw water tank might overflow, often causing a severe damage to the system. This might occur due to the PLC1 downtime τ, during which, the control command (i.e. "ON") computed by PLC1 may not change. Similarly, if the actuator receives the command "OFF" from PLC1 for an arbitrarily long duration, then

the water tank may underflow. Such a phenomenon will still affect the system dynamics. This is because tanks from other processes may expect raw water from this underflow tank. In the context of SWaT, the *tolerability* of PLC1 downtime τ (cf. Eq. (4)) depends on the physical state of the water tank (i.e. water level) and the computed control commands (i.e. ON or OFF) by PLC1. In the next section, we will formally introduce this notion of tolerance, as termed *physical-state resiliency*, for a typical CPS.

To formally model the physical-state resiliency, we will take a control-theoretic approach. For the sake of simplifying the presentation, we will assume that the dynamics of a typical CPS, without considering the noise and disturbance on the controller, is modeled via a linear-time invariant. This is formally captured as follows (cf. Figure 1):

$$x_{t+1} = Ax_t + Bu_t \tag{5}$$

$$y_t = Cx_t \tag{6}$$

where $t \in \mathbb{N}$ captures the index of discretized time domain. $x_t \in \mathbb{R}^k$ is the state vector of the physical plant at time t, $u_t \in \mathbb{R}^m$ is the control command vector at time t and $y_t \in \mathbb{R}^k$ is the measured output vector from sensors at time t. $A \in \mathbb{R}^{k\times k}$ is the state matrix, $B \in \mathbb{R}^{k\times m}$ is the control matrix and $C \in \mathbb{R}^{k\times k}$ is the output matrix.

We now consider a duration $\tau \in \mathbb{R}$ for the PLC downtime. To check the tolerance of τ, we need to validate the physical state vector x_t at any discretized time index t. To this end, we first assume an upper-bound $\omega \in \mathbb{R}^k$ on the physical state vector x_t at an arbitrary time t. Therefore, to satisfy the physical-state resiliency, x_t must not exceed the threshold ω. In a similar fashion, we define a lower-bound $\theta \in \mathbb{R}^k$ on the physical state vector x_t.

With the PLC downtime τ, we revisit Eq. (5) and the state estimation is refined as follows:

$$x'_{t+1} = Ax_t + Bu_{t-1}[\![t, t+\tau]\!] \tag{7}$$

where $x'_{t+1} \in \mathbb{R}^k$ is the estimated state vector at time $t+1$ and the PLCs were down for a maximum duration τ. The notation $u_{t-1}[\![t, t+\tau]\!]$ captures that the control command u_{t-1} was active for a time interval $[t, t+\tau]$ due to the PLC downtime for a duration of τ. In Eq. (7), we assume, without loss of generality, that u_{t-1} is the last control command received from the PLC before its downtime.

Given the foundation introduced in the preceding paragraphs, we say that a typical CPS (cf. Figure 1) satisfies physical-state resiliency if and only if the following condition holds at an arbitrary time index t:

$$\theta \leq x'_{t+1} \leq \omega$$

$$\theta \leq Ax_t + Bu_{t-1}[\![t, t+\tau]\!] \leq \omega \tag{8}$$

Figure 4 illustrates three representative scenarios to show the consequence of Eq. (8). If the downtime $\tau_1 = 0$, then u_t (i.e. control command at time t) is correctly computed and $x'_{t+1} = x_{t+1}$. If the downtime $\tau_2 \in (1, 2]$, then the control command u_t will be the same as u_{t-1}. Consequently, x'_{t+1} is unlikely to be equal to x_{t+1}. Finally, when downtime $\tau_3 > 2$, the control command vector u_{t+i} for $i \geq 0$ will be the same as u_{t-1}. As a result, the estimated state vectors x'_{t+j} for $j \geq 1$ will unlikely to be identical to x_{t+j}.

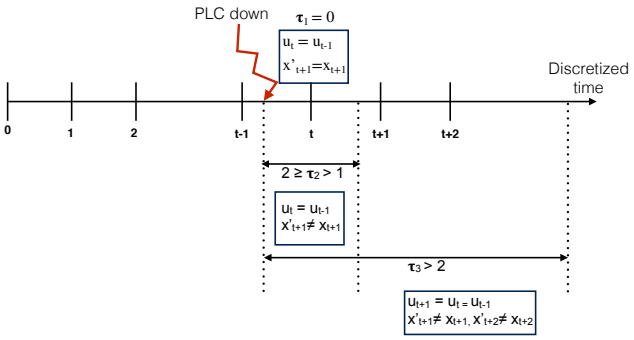


Figure 4: Illustrating the impact of PLC downtime

4. CIMA: Countering Illegal Memory Accesses

CIMA is a mitigation technique designed to counter illegal memory accesses at runtime. We first outline a high-level overview and the key ingredients of our approach. Later in this section, we discuss the specific implementation traits in more detail.

4.1. Overview of CIMA

Objective

CIMA follows a proactive approach to mitigate memory-safety attacks and thereby enhancing system availability and physical-state resilience in CPS. The key insight behind CIMA is to prevent any operation that attempts to illegally access memory. We accomplish this by skipping (i.e. not executing) those instructions that attempt to access memory illegally. For example, consider the exploitation of a memory-safety attack shown in Figure 3. With our CIMA approach, the attack will be ceased at step 3 (i.e. "Trigger a memory-safety violation") of the exploitation process.

Workflow of CIMA

CIMA systematically manipulates the compiler-generated control flow graph (CFG) to accomplish its objective, i.e., to skip illegal memory accesses at runtime. CFG of a program, specifically at GIMPLE [44] intermediate representation (IR) level, is represented by a set of basic blocks having incoming and outgoing edges. A basic block [45] is a straight-line sequence of code with only one entry point and only one exit, but it can have multiple predecessors and successors. CIMA works on a common intermediate representation, i.e., GIMPLE IR, of the underlying code, thus making our approach applicable for multiple high-level programming languages and low-level target architectures.

To manipulate the CFG, CIMA needs to instrument the set of memory-access operations in such a fashion that they trigger memory-safety violations at runtime. To this end, CIMA

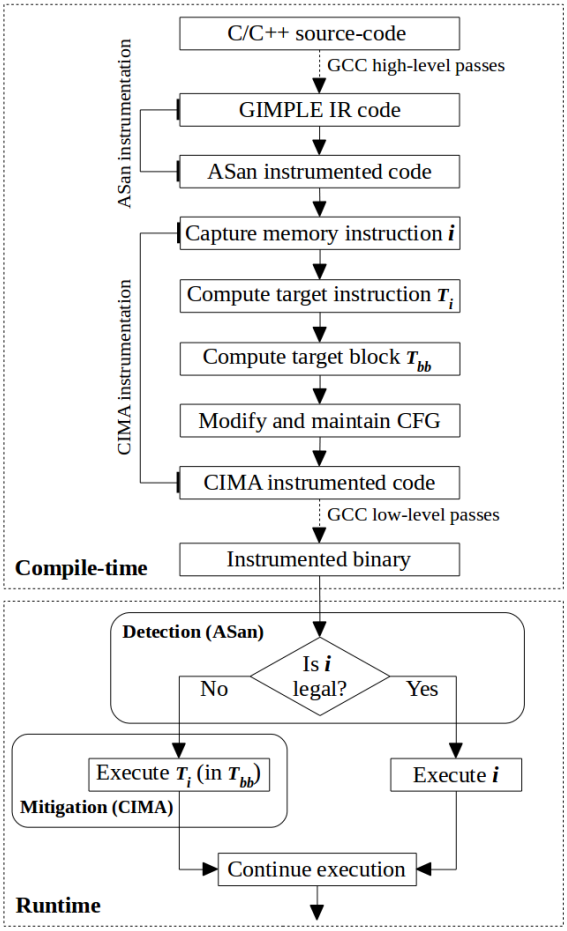


Figure 5: A high-level architecture of CIMA

leverages off-the-shelf technologies, namely address sanitizers (ASan). A high-level workflow of our entire approach appears in Figure 5. The implementation of CIMA replaces the ineffective mitigation strategy (i.e. system abort) of ASan with an effective, yet lightweight scheme. Such a scheme allows the program control flow to jump to the immediately next instruction that does not exhibit illegal memory access. In such a fashion, CIMA not only ensures high accuracy of memory-safety attack mitigation, but also provides confidence in system availability and resilience. In the following section, we describe our CIMA approach in detail.

4.2. Approach of CIMA

An outline of our CIMA approach appears in Algorithm 2. As input, CIMA takes a function (in the GIMPLE format) instrumented by ASan. As output, CIMA generates a GIMPLE function with memory-safety mitigation enabled. We note that the modification is directly injected in the compiler workflow. To this end, CIMA modifies the internal data structures of GCC to reflect the changes of control flow. Specifically, CIMA manipulates the intermediate representation, which is often derived in the static-single-assignment (SSA) form. To this end, CIMA also ensures that the SSA form is maintained during the manipulation, so as not to disrupt the compiler workflow. In the following, we elaborate the four crucial steps of CIMA in detail and discuss the specific choices taken during its implementation.

Algorithm 2: CIMA's approach to ensure memory safety

Input: Function *fun*() with ASan Instrumentation
Output: Function *funCIMA*() with CIMA-enabled memory safety

```
1  forall the basic block bb in fun() do
2      forall the instruction i instrumented by ASan in bb do
3          Let check_bb holds the memory-safety check condition
           check_i for instruction i in bb
4          Let abort_bb be the basic block where control reaches to
           when instruction i is illegal
5          Find target instruction T_i for instruction i
6          if ( i and T_i reside in the same basic block bb ){
7              temp_bb := succ(bb)
8              /* split basic block */
9              Split bb into basic blocks i_bb and T_bb
10             i_bb holds instructions of bb up to i
11             T_bb holds instructions from T_i up to the end of bb
12             /* Modify control flow */
13             /* succ(bb) captures successor of bb */
14             succ(i_bb) := {T_bb}
15             succ(check_bb) := {i_bb, T_bb}
16             succ(T_bb) := temp_bb
17         }
18         else{
19             /* Modify control flow */
20             Let T_i be in basic block T_bb
21             succ(check_bb) := (succ(check_bb) \ {abort_bb}) ∪ {T_bb}
22         }
23     end
24 end
```

Capture memory access instructions: CIMA validates memory accesses at runtime to detect and mitigate illegal memory access instructions. To achieve this, CIMA combines a compile-time code instrumentation and a runtime validity check techniques. The code instrumentation includes instrumenting memory addresses (via ASan) and memory access instructions (via CIMA). As discussed in Section 2.2, the memory address instrumentation creates the poisoned *redzones* around stack, heap and global variables. Since these redzones are inaccessible by the running program, any memory instruction, attempting to access them at runtime will be detected as an illegal instruction.

To mitigate the potential illegal instructions, CIMA instruments memory access instructions at compile-time. To this end, CIMA captures memory access instructions from the ASan instrumented code. These instructions serve as the potential candidates for illegal instructions at runtime.

Compute target instruction: To prevent the execution of an illegal memory access instruction i, CIMA computes its corresponding target instruction T_i. We note that the set of potentially illegal instructions are computed via the technique as explained in the preceding paragraph. *The target instruction T_i for an illegal instruction i is the instruction that will be executed right after i is bypassed.* The compile-time analysis of CIMA

generates an instrumented binary in such a fashion that if an illegal instruction i is detected at runtime, then i is bypassed and control reaches target instruction T_i. If T_i is detected as illegal too, then the successor of T_i will be executed. This process continues until an instruction is found without illegal memory access at runtime.

Computing the target instruction is a critical step in our CIMA approach. The target instruction T_i is computed as the successor of the memory access instruction i in the CFG. We note that a potentially illegal instruction must access memory, as the objective of CIMA is to mitigate memory-safety attacks. At the GIMPLE IR-level, any memory access instruction has a single successor. Such a successor can either be the next instruction in the same basic block (see Figure 6) or the first instruction of the successor basic block (see Figure 7). Since CIMA works at the GIMPLE IR-level, it can identify the target instruction T_i for any instruction i by walking through the static control flow graph.

We modify the control flow graph in such a fashion that the execution is diverted to T_i (i.e. the successor of i) at runtime when instruction i is detected to be illegal at runtime.

Computing the target basic block: From the discussion in the preceding paragraph, we note that the execution of memory access instruction i is conditional (depending on whether it is detected as illegal at runtime). In the case where the instruction exhibits an illegal memory access, a jump to the respective target instruction T_i is carried out. However, it is not possible to simply divert the execution flow to the target instruction T_i. This needs to be accomplished via a systematic modification of the original program CFG at compile-time. However, the modification of the CFG depends on the location of the illegal and target instructions. There are two scenarios in this regard.

Scenario 1 is when the illegal instruction i and its respective target instruction T_i reside in the same basic block (say bb). In this case, since the execution of i is conditional, it is always the first instruction of its holding basic block whereas the target instruction T_i appears as the next instruction in the basic block bb (see Figure 6). However, it is not possible to make a conditional jump to T_i within the same basic block bb, as this breaks the structure of the control flow graph. Thus, to be able to make a conditional jump to T_i, a target basic block (say T_{bb}), that contains T_i as its head instruction, is created. In particular, we split the basic block bb into two basic blocks – i_{bb} and T_{bb} (cf. Algorithm 2, Lines 9 – 11). i_{bb} holds the potentially illegal instruction i and T_{bb} holds the instructions of bb following i, i.e., starting from the target instruction T_i and up to the last instruction of bb. As such, control jumps to T_{bb} if instruction i is detected as illegal at runtime.

Scenario 2 is when the illegal instruction i and its respective target instruction T_i are located in different basic blocks in the original CFG (see Figure 7), then there is no need to split any basic block. In such a case, the target basic block T_{bb} will be the basic block holding T_i as its first instruction. Therefore, CIMA diverts the control flow to T_{bb}, should i exhibits an illegal memory access at runtime.

Modify and maintain CFG: CIMA ensures the diversion of control flow of the program whenever an illegal memory access is detected. To accomplish such a twisted control flow, CIMA directly modifies and maintains the CFG (cf. Algorithm 2, Lines (14 – 16) and (21)), as explained in the preceding paragraph. Figure 6 and 7 illustrate excerpts of control flow graphs that are relevant to the modification performed by CIMA in scenario 1 and 2, respectively. In particular, the two figures demonstrate how the control flow is systematically manipulated to guarantee the system availability, while still mitigating memory-safety attacks.

Figure 6 and 7 also illustrate the change in CFG when the program is compiled without and with memory-safety. In the former case, the figures demonstrate how memory-safety attacks (such as code injection and code reuse attacks) divert the execution flow of the memory access instruction i to the attacker injected code or system modules to further synthesis their exploitation. In the later case, the figures demonstrate how the ASan and CIMA instrumented program systematically modifies the CFG to detect and mitigate the attacker's attempt to illegally divert the execution flow of instruction i.

In Figure 6 and 7, $check_i$ is the inserted conditional check by ASan to identify illegal memory accesses in instruction i and $check_{bb}$ is the basic block holding $check_i$. Similarly, $abort_{bb}$ is a basic block in the ASan instrumented code to which control jumps to if i exhibits illegal memory access. However, $abort_{bb}$ is excluded in the CIMA's modified CFG as control jumps to the computed target basic block (T_{bb}) instead if i turns out to be illegal.

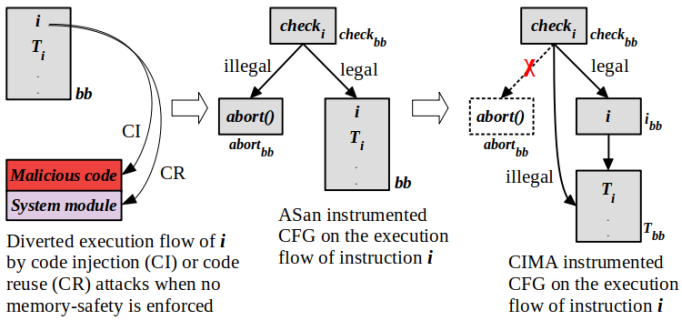


Figure 6: Illustrating the change of CFG in scenario 1 (i.e. i and T_i reside in same basic block) involving three cases: attacker diverted CFG (without memory-safety), ASan instrumented CFG (preventing exploitation but crashing execution) and CIMA instrumented CFG (preventing exploitation and continuing execution).

4.3. Illustrative Example

Using CIMA, we detected and mitigated a global buffer overflow vulnerability on the firmware of OpenPLC controller (we reported the vulnerability and a CVE ID has already been assigned to it [46, 47]). A code fragment relevant to the vulnerability is shown in Program 3. In Line 3, a buffer "*int_memory[]*" (with a buffer size of 1024) is declared in the "*glue_generator.cpp*" file. This buffer is also used in the "*modbus.cpp*" file, enclosed within a *for* loop (Lines 19 – 30) under the "*mapUnusedIO()*" function. In the loop, the memory write operation "*int_memory[i] = &mb_holding_regs[i]*" (Line 27)

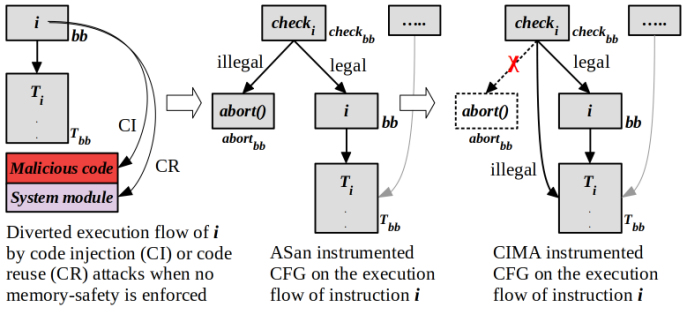


Figure 7: Illustrating the change of CFG in scenario 2 (i.e. i and T_i reside in different basic blocks) involving three cases: attacker diverted CFG (without memory-safety), ASan instrumented CFG (preventing exploitation but aborting execution) and CIMA instrumented CFG (preventing exploitation and continuing execution)

writes data to the buffer. However, due to a coding error, this operation exhibits a memory-safety violation. Such a violation occurs in the 1024^{th} iteration when the operation attempts to write data beyond the buffer limit. CIMA successfully mitigates such memory-safety violation. This was possible as the illegal memory access operation was bypassed in each iteration starting from the 1024^{th} iteration of the loop.

4.4. Validating CIMA

CIMA is designed to alter the original and potentially vulnerable program to prevent run-time exploitation. As such, it modifies the original program semantics. In the following we discuss potential consequences of changes in the original program's low-level semantics.

First, note that it is fair to assume that inputs that would exploit a memory-safety vulnerability are outside the intended original program semantics, and that their behaviour is thus not specified. As such, CIMA does not modify the intended program semantics for the set of sequences of inputs that do not trigger a memory-safety violation. For all other inputs, it would be acceptable to just interrupt program execution to prevent exploitation (as in the case of many memory-safe compilers, that will throw some kind of run-time exception and crash). In our context, however, we want to prevent the program from crashing in order to guarantee higher levels of availability and thus the interesting question is whether we can give guarantees in this sense.

We distinguish then the following three fundamental cases, when a modified program is run with a malicious (memory-safety violating) input:

1. The modified program is able to finish the PLC scan cycle before the real-time deadline. In this case, we have successfully prevented exploitation while preserving the system's availability. According to our experiments, the behaviour exhibited by the PLC will still be correct with respect to the specified controller behaviour in most cases. This correctness can be nevertheless verified at runtime by an orthogonal mechanism that monitors input sensor values and the corresponding decision made by the PLC, judging from the overall system behaviour and network or historian values, for instance using physical invariants [37] or process models [38, 39, 40].

Program 3: The OpenPLC vulnerability

```
1  //————————//glue_generator.cpp————————————————
2  #define BUFFER_SIZE 1024
3  IEC_UINT *int_memory[BUFFER_SIZE];
4  IEC_UINT *int_output[BUFFER_SIZE];
5  .
6  .
7  //————————//modbus.cpp————————————————————————
8  #define MIN_16B_RANGE 1024
9  #define MAX_16B_RANGE 2047
10 #define MAX_HOLD_REGS 8192
11 IEC_UINT mb_holding_regs[MAX_HOLD_REGS];
12 .
13 .
14 /* This function sets the internal OpenPLC buffers to */
15 /* point to valid positions on the Modbus buffer */
16 void mapUnusedIO() {
17     .
18     .
19     for( int i = 0; i <= MAX_16B_RANGE; i++ ){
20         if ( i <MIN_16B_RANGE ){
21             if ( int_output[i] == NULL ){
22                 int_output[i] = &mb_holding_regs[i];
23             }
24         }
25         if ( i >= MIN_16B_RANGE && i <= MAX_16B_RANGE
           ){
26             if ( int_memory[i - MIN_16B_RANGE] == NULL ){
27                 int_memory[i] = &mb_holding_regs[i];
28             }
29         }
30     }
31 }
```

2. The modified program is not able to finish the PLC scan cycle before the real-time deadline. This situation can be caused by either the modified program becoming non-terminating, or because the modified terminates, but because of the modification, ends up executing prohibitively slow. It is possible to construct programs where this situation could arise. For example, Program 4 shows a case where the loop bound (Line 5) is dependent on an attacker controlled input x. We note that the value of x determines whether the memory read instruction *arr[x]* is legal or not. CIMA skips the access to *arr[x]* should it turns out to be illegal. This could lead to premature abortion of the loop (e.g. if *arr[x]* is detected as illegal before the loop starts). Consider a different scenario when the the loop counter i is incremented by *arr[z]* (cf. Program 4, Line 8). Assuming z can be controlled by the attacker, *arr[z]* might manifest illegal memory access. Thus, CIMA bypasses the respective memory access and the value of i remains unchanged. This leads to an execution that never terminates.

These corner cases could occur when a false data injection attack manages to control the loop bound and loop counter of a *for* loop. However, we believe such cases are rather pathological and that in practice it will be very hard for attackers to tailor their inputs to cause this behaviour. Providing *loop bound* and

loop counter values via attacker controlled memory accesses is also certainly considered as a *bad coding practice*. In fact, the occurrence of such corner cases could result in an undesirable outcome (e.g. program crash) even in the absence of CIMA. Also, it is probably easier for attackers to resort to more generic DoS attacks such as flooding than crafting such inputs, although more work in this direction is left for the future. In this situation, to ensure availability, it would be necessary to devise mechanisms to either roll-back to a safe-state and ignore malicious inputs or other countermeasures. Note that however, it is fairly easy to detect such cases by considering the normal execution profile of the PLC (which is usually much faster than the real-time deadline) and raising appropriate alarms. In particular, false data injection attacks can be also prevented via orthogonal means [48].

3. When a variable is dependent on the output of a memory access instruction. For example, in Program 4, Line 7, variable *y* is dependent on the output of memory access instruction *arr[i]*. If the access *arr[i]* is found to be illegal, CIMA skips it and the variable assignment, i.e., *y = arr[i]*, will be also skipped as a side-effect. In this case, *y* preserves the last "legally" assigned value to it. Otherwise, variable *y* preserves its initial value if the access *arr[i]* is detected as illegal at the beginning (i.e. at index $i = 0$).

We also take some careful considerations when implementing CIMA. Concretely, we ensure that the implementation of CIMA is not affected by lower level compiler optimizations. CIMA does not also have any impact on the work flow of the compiler back-end. Concurrently, the modified compiler flow does not influence the execution of programs without any memory access violation.

In sum, we believe that CIMA can significantly increase the resilience of CPS against attacks that are focused on memory-safety exploitation. Since such countermeasures are not common in CPS today, this will prevent many realistic attacks. If attackers are aware of countermeasures such as CIMA, it is thinkable that they would also attempt to craft more sophisticated attacks against availability, but this is left for future study.

Program 4: Attacker controlled loop bound and counter

```
1  main() {
2      int arr[100], i, x, y, z;
3      printf("Enter the value of x and z:");
4      scanf("%d %d", &x, &z);
5      for( i=0; i <arr[x]; ){
6          // Do something
7          y = arr[i]; //Variable assignment
8          i += arr[z]; //Incrementing the loop counter
9      }
10     return 0;
11 }
```

4.5. Implementation challenges

ASan instruments memory addresses to detect illegal memory access instructions and it aborts the program upon detection of illegal memory accesses, which affects availability of the system. In our CIMA approach, we make the observation that we can react to such violations by bypassing the illegal instructions, and thus favoring availability of the system. However, unlike aborting the vulnerable program (which can be achieved by simply calling the *abort()* library function), skipping only illegal memory access instructions and keeping the rest of the execution alive is not a straightforward task. As such, we face several technical challenges to realize the aforementioned intuition behind CIMA. Firstly, it is infeasible (in general) to statically compute the exact set of illegal memory accesses in a program. Consequently, a fully static approach, which modifies the program to eliminate the illegal instructions from the program, is unlikely to be effective. Moreover, such an approach will inevitably face scalability bottlenecks due to its heavy reliance on sophisticated program analysis. Secondly, even if the illegal instructions are identified during execution, it is challenging to bypass the manifestation of illegal memory access. This is because, such a strategy demands full control to manipulate the normal flow of program execution. Finally, to bypass the execution of certain instructions, we need modifications to the control flow of the program, which involves the manipulation of program control flow over several compilation passes.

To overcome the technical challenges, we implement CIMA by modifying the GCC middle-end, i.e., GIMPLE IR level, where the CFG is represented by a set of basic blocks (consisting of a sequence of instructions). The compiler middle-end is modified in such a way that the modified compiler produces a new CFG (for a given program) according to the modifications outlined in Figures 6 and 7. This newly formed CFG will then aid to skip instructions, which exhibit illegal memory accesses, via a conditional jump.

In summary, unlike ASan, CIMA exclusively instruments memory access instructions to bypass those instructions exhibiting memory access violations. The rest of the execution, nevertheless, continues without interruption. To the best of our knowledge, there is no any existing strategy that skips illegal memory access instructions without aborting or interrupting execution of the victim program.

5. Experimental Design

5.1. SWaT

SWaT [49] is a fully operational water purification testbed for research in the design of secure cyber-physical systems. It produces five gallons/minute of doubly filtered water. In the following, we discuss some salient features and design considerations of SWaT.

5.1.1. Purification process

The entire water purification process is carried out by six distinct, yet co-operative, sub-processes. Each sub-process is controlled by an independent PLC (indexed from PLC1 through PLC6). Specifically, PLC1 controls the first sub-process, i.e., the inflow of water from external supply to a raw water tank, by

opening and closing the motorized valve connected with the inlet pipe to the tank. PLC2 controls the chemical dosing process, e.g., water chlorination, where appropriate amount of chlorine and other chemicals are added to the raw water. PLC3 controls the ultrafiltration (UF) process. PLC4 controls the dechlorination process where any free chlorine is removed from the water before it is sent to the next stage. PLC5 controls the reverse osmosis (RO) process where the dechlorinated water is passed through a two-stage RO filtration unit. The filtered water from the RO unit is sent in the permeate tank, where the recycled water is stored, and the rejected water is sent to the UF backwash tank. In the final stage, PLC6 controls the cleaning of the membranes in the UF backwash tank by turning on and off the UF backwash pump. The overall purification process of SWaT is shown in Figure 8 (more details can be found in [50]).

5.1.2. Components and specifications

The design of SWaT consists of the following components and system specifications:

- **PLCs**: six redundancy real-world PLCs (Allen Bradley PLCs) to control the entire water purification process. The PLCs communicate one another or with the SCADA system through EtherNet/IP or common industrial protocol (CIP).
- **Remote input/output (RIO)**: SWaT also consists of remote input/output terminals containing digital inputs (DI), digital outputs (DO) and analog inputs (AI) for each PLC. The RIO of SWaT consists of 32 DI (water level and pressure switches), 13 AI (flow rate, water pressure and water level sensors), and 16 DO (actuators such as pumps and motorized valves).
- **PLC program**: SWaT has a complex PLC program (control software) written in ladder logic. It comprises various instructions such as boolean operators, arithmetic operators, comparison operators, conditional statements, counters, timers, contacts, and coils (see the full list on Table 1). The most complex PLC of SWaT (i.e. PLC2) has a PLC program containing 129 instructions.
- **SCADA system**: a touch panel is mounted to SWaT for providing users a local system supervisory, monitoring and control. It also displays state information of plants, sensors, actuators and operational status of PLCs to users.
- **Operation management**: consisting of historian server (for storing all operating data, alarm history and events) and engineering workstation (designed to provide all necessary control graphics).
- **Real-time constraint**: the cycle time or real-time constraint of SWaT is 10ms. The notion of real-time constraint (in the context of CPS) is discussed in detail on Section 3.2.
- **Communication frequency**: the six PLCs communicate each other and with the SCADA system depending on operational conditions. Considering the most complex PLC (with 129 instructions), it send as high as 382 packets per second to its most active peer or as low as three packets per second to another peer. Considering connections with all devices in SWaT, we estimate that the most complex PLC has a send over receive request ratio of 1000 packets per second.

Concurrently, SWaT is based on closed-source and proprietary Allen Bradely PLCs. Hence, it is not possible to directly modify the firmware of these PLCs and to enforce memory-safety solutions. To alleviate this problem in our experimental evaluation, an open platform, named Open-SWaT, was designed.

5.2. Open-SWaT

Open-SWaT is a mini CPS we designed using OpenPLC controller [51] – an open source PLC designed for industrial control and cyber-physical systems. A high-level architecture of Open-SWaT is shown in Figure 9. The main purpose of designing Open-SWaT was to employ our CIMA approach on a realistic CPS. With Open-SWaT, we reproduce features and operational behaviours of SWaT. In particular, we reproduce the main factors that have substantial effect on the scan time and MSO of PLCs. Some salient features of the Open-SWaT design are discussed as follows.

1. **PLCs**: The PLCs of Open-SWaT are designed using OpenPLC controller that runs on a Raspberry PI using Linux operating system. To replicate the hardware specifications of the PLCs in SWaT, we configured 200MHz fixed CPU frequency and 2Mb user memory for the PLCs used in Open-SWaT.
2. **Remote Input/Output (RIO)**: Arduino Mega has been used as the RIO terminal in Open-SWaT. It has an AVR-based processor with a clock rate of 16MHz. It consists of 86 I/O pins that can be directly connected to different I/O devices. To replicate the number of input and outputs in SWaT, we used 32 digital inputs (DI) (switches, push-buttons and scripts), 13 analog inputs (AI) (ultrasonic and temperature sensors) and 16 digital outputs (DO) (light emitter diodes or LEDs, in short).
3. **PLC program**: We designed a control program written in ladder logic. It has similar complexity with the one in SWaT. A sample of the logic diagram is shown in Figure 10. The control program consists of several types of instructions such as 1) *logical:* AND, OR, NOT, SR (set-reset latch); 2) *arithmetic*: addition (ADD), multiplication (MUL); 3) *comparisons:* equal-to (EQ), greater-than (GT), less-than (LT), less-than-or-equal (LE); 4) *counters:* up-counter (CTU), turn-on timer (TON), turn-off timer (TOF); 5) *contacts:* normally-open (NO) and normally-closed (NC); and 6) *coils:* normally-open (NO) and normally-closed (NC). The overall control program consists of 129 instructions in total; details are shown on Table 1.
4. **Communication frequency**: The communication architecture of Open-SWaT (as illustrated in Figure 9) consists of analogous communicating components with that

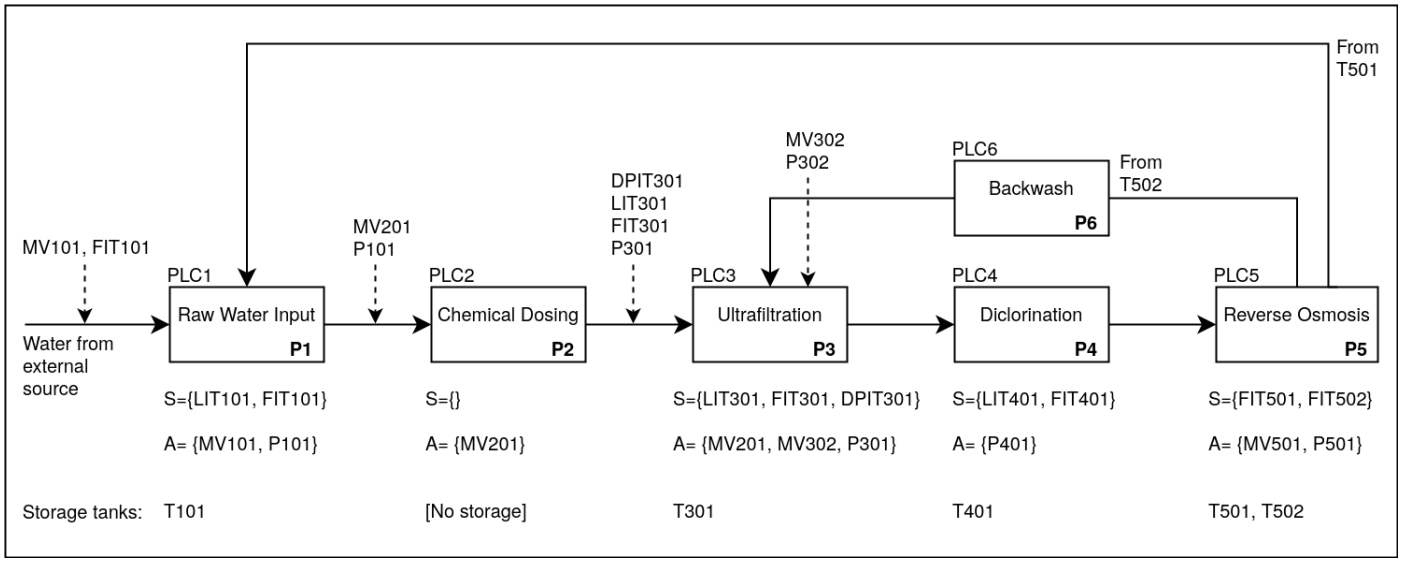


Figure 8: Overview of the water purification process of SWaT
Definition of the acronyms: S = Sensor, A = Actuator, T = Tank, P = Process, MV = Motorized Valve, LIT = Level Indicator Transmitter, FIT = Flow Indicator Transmitter, DPIT = Differential Pressure Indicator Transmitter.

of SWaT. Open-SWaT uses both types of modbus protocols – modbus TCP (for wired or wireless communication) and modbus RTU (for serial communication). The communication among PLCs is via modbus TCP or modbus RTU whereas the communication between PLCs and the SCADA system is via modbus TCP. Frequency of communication among PLCs and the SCADA system is similar to that in SWaT. The communication between PLCs and Arduino is via the USB serial communication. The communication frequency between Arduino and sensors is 100Hz.

5. **Real-time constraint**: Since the cycle-time (real-time constraint) of SWaT is 10ms, we also set 10ms cycle time to each PLC in Open-SWaT.
6. **SCADA system**: We use ScadaBR [52], a full SCADA system consisting of web-based HMI, for Open-SWaT.

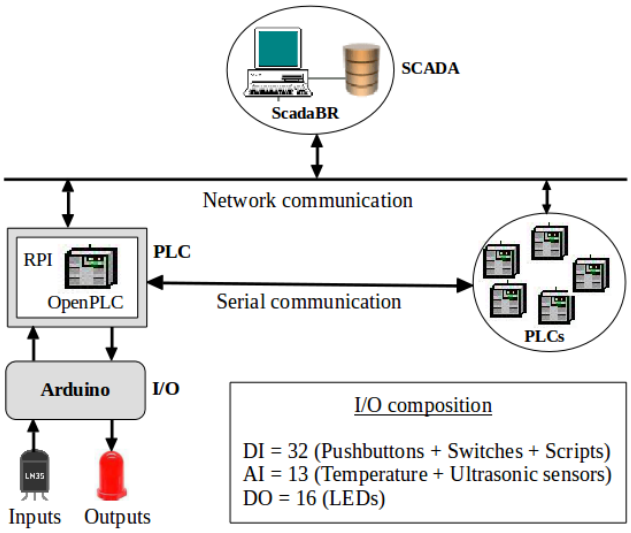


Figure 9: Architecture of Open-SWaT

5.3. *SecUTS*

The Secure Urban Transportation System (SecUTS) is a CPS testbed designed to research on the security of a Metro SCADA system. The Metro SCADA system [53] comprises an *Integrated Supervisory Control System (ISCS)* and a *train signaling system*. ISCS integrates localized and centralized control and supervision of mechanical and electrical subsystems located at remote tunnels, depots, power substations and passenger stations. The entire Metro system can be remotely communicated, monitored, and controlled from the operation control center via the communication network. On the other hand, the signaling system facilitates communications between train-borne and track-side controllers. It also controls track-side equipments and train position localization. Modbus is used as a communication protocol among the devices in the ISCS. A detailed account of the Metro SCADA can be found on [53].

The SecUTS testbed provides facilities to examine several types of cyber attacks, such as message replay, forged message and memory-safety attacks, in the ISCS system and enforce proper countermeasures against such attacks. However, the SecUTS testbed is also based on closed-source proprietary Siemens PLCs, hence we cannot directly enforce CIMA to these PLCs to detect and mitigate memory-safety attacks. Consequently, we similarly designed **Open-SecUTS** testbed (by mimicking SecUTS) using OpenPLC controller. It consists of 6 DI (emergency and control buttons) and 9 DO (tunnel and station lightings, ventilation and alarms) and a cycle time of 30ms. Subsequently, we enforced CIMA to Open-SecUTS and evaluated its practical applicability in a Metro SCADA system (cf. Section 6).

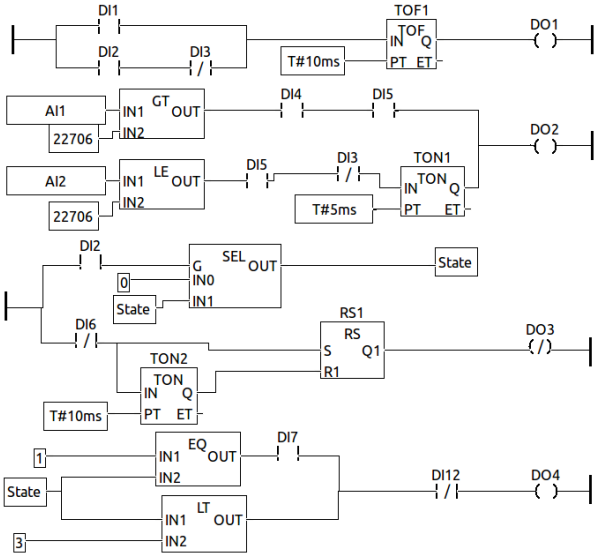


Figure 10: Sample PLC program in ladder diagram

Table 1: Complexity of the SWaT PLC program

Instruction(s)	Type	count
AND	Logical	17
OR	Logical	14
NOT	Logical	5
SR	Logical	1
ADD	Arithmetic	1
MUL	Arithmetic	2
EQ , GT	Comparison	6
LT , LE	Comparison	4
TON	Timers	3
TOF	Timers	9
CTU	Counters	1
SEL, MAX	Selection	2
NO	Contacts	38
NC	Contacts	3
NO	Coils	21
NC	Coils	2
Total		129

6. Evaluation

We discuss a detailed evaluation of our CIMA approach on Open-SWaT and Open-SecUTS. Subsequently, we discuss the experimental results to figure out whether our proposed approach is accurate enough to detect and mitigate memory-safety violations. We also discuss the efficiency of our approach in the context of CPS environment. In brief, we evaluate the proposed approach along four dimensions: *1) Security guarantees* – detection and mitigation accuracy of the proposed approach *2) Performance* – tolerability of the runtime overhead of the proposed security measure in CPS environment, *3) Resilience* – its capability to ensure system availability and maintain physical-state resiliency in CPS even in the presence of memory-safety attacks, and *4) its Memory usage overheads*.

6.1. Security guarantees

To stress test the accuracy of our approach, we have evaluated CIMA against a wide-range of memory-safety vulnerabilities. This is to explore the accuracy of mitigating memory-safety vulnerabilities by our CIMA approach. As our CIMA approach is built on top of ASan, it is crucial that ASan *detects* a wide-range of memory-safety vulnerabilities accurately. According to the original results published for ASan [9], it detects memory-safety violations with high-accuracy – without false positives for vulnerabilities such as (stack, heap and global) buffer under/overflows, use-after-free errors (dangling pointers), use-after-return errors, initialization order bugs and memory leaks. Only rare false negatives may appear for global buffer overflow and use-after-free vulnerabilities due to the exceptions discussed in [9].

CIMA effectively *mitigates* memory-safety violations, given that such a violation is detected by ASan at runtime. Therefore, the mitigation accuracy of our CIMA approach is exactly the same as the detection accuracy of ASan.

As discussed in detail on Section 4.2, we discovered a global buffer overflow vulnerability in the OpenPLC firmware [46]. The vulnerability was successfully mitigated by our CIMA approach. Besides, throughout our evaluation, we did not discover any false positives or negatives in mitigating all the memory-safety violations detected by ASan.

6.2. Efficiency

According to the original article published for ASan [9], the average MSO of ASan is 73%. However, all measurements were taken on benchmarks different from ours and more importantly, in a non-CPS environment. With our CPS environment integrated in the Open-SWaT and Open-SecUTS, the average performance overhead induced by ASan is 53.46% and 50.4%, respectively. Additionally, our proposed CIMA approach induces 8.06% and 6.53% runtime overheads on Open-SWaT and Open-SecUTS, respectively. Thus, the overall runtime overhead of our security measure is 61.52% (for Open-SWaT) and 56.93% (for Open-SecUTS). A more detailed performance report, including the performance overhead of each PLC operation in both testbeds, is illustrated on Table 2 and 3.

It is crucial to check whether the induced overhead by ASan and CIMA ($\hat{T}_s$) is tolerable in a CPS environment. To this end, we evaluate if this overhead respects the *real-time constraints* of SWaT and SecUTS. For instance, consider the tolerability in average-case scenario. We observe that our proposed approach satisfies the condition of tolerability, as defined in Eq. (2). In particular, from Table 2, $mean(\hat{T}_s)$ = 441.72µs, and T_c = 10ms; and from Table 3, $mean(\hat{T}_s)$ = 398.39µs, and T_c = 30ms. Consequently, Eq. (2) is satisfied and the overhead induced by our CIMA approach is both tolerable in SWaT and SecUTS.

Similarly, considering the worst-case scenario, we evaluate if Eq. (3) is satisfied. From Table 2, $max(\hat{T}_s)$ = 3167.15µs, and T_c = 10ms; and from Table 3, $max(\hat{T}_s)$ = 2506.39µs, and T_c =

Table 2: Memory-safety overheads for the Open-SWaT Testbed

Operations	Number of cycles	Network devices	CPU speed (in MHz)	Original (T_s)		ASan(T'_s)				ASan + CIMA ($\hat{T}_s$)			
				$mean(T_s)$ (in µs)	$max(T_s)$ (in µs)	$mean(T'_s)$ (in µs)	$max(T'_s)$ (in µs)	MSO (in µs)	MSO (in %)	$mean(\hat{T}_s)$ (in µs)	$max(\hat{T}_s)$ (in µs)	MSO (in µs)	MSO (in %)
Input scan	50000	6	200	59.38	788.12	118.44	1132.32	59.06	99.46	122.86	1151.35	63.48	106.9
Program exec.	50000	6	200	69.09	611.82	115.88	720.36	46.79	67.72	118.97	802.18	49.88	72.2
Output update	50000	6	200	145.01	981.09	185.37	1125.45	40.36	27.83	199.89	1213.62	54.88	37.85
Full scan time	50000	6	200	273.48	2381.03	419.69	2978.13	146.21	53.46	441.72	3167.15	168.24	61.52

Table 3: Memory-safety overheads for the Open-SecUTS Testbed

Operations	Number of cycles	Network devices	CPU speed (in MHz)	Original (T_s)		ASan(T'_s)				ASan + CIMA ($\hat{T}_s$)			
				$mean(T_s)$ (in µs)	$max(T_s)$ (in µs)	$mean(T'_s)$ (in µs)	$max(T'_s)$ (in µs)	MSO (in µs)	MSO (in %)	$mean(\hat{T}_s)$ (in µs)	$max(\hat{T}_s)$ (in µs)	MSO (in µs)	MSO (in %)
Input scan	50000	1	200	59.84	739.94	114.88	902.01	55.04	91.98	115.07	906.09	55.23	92.3
Program exec.	50000	1	200	48.56	488.38	91.36	443.61	42.8	88.14	104.41	676.19	55.85	115.01
Output update	50000	1	200	145.47	850.62	175.59	1045.34	30.12	20.71	178.91	924.11	33.44	22.99
Full scan time	50000	1	200	253.87	2078.94	381.83	2390.96	127.96	50.4	398.39	2506.39	144.52	56.93

30ms. It is still tolerable, thus the proposed security measure largely meets the real-time constraints of SWaT and SecUTS in both scenarios (See the tolerability gap for SWaT and SecUTS (in the worst-case scenario) in Figure 11 and 12, respectively). Therefore, despite high security guarantees provided by CIMA, its overhead is still tolerable in a CPS environment.

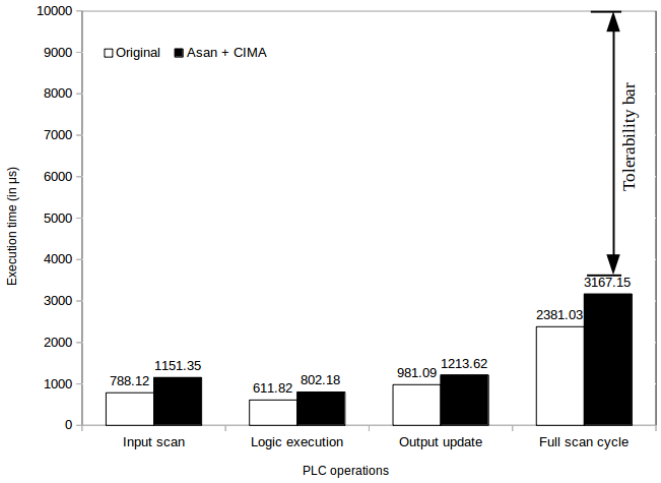


Figure 11: Tolerability of the worst-case overhead for Open-SWaT testbed

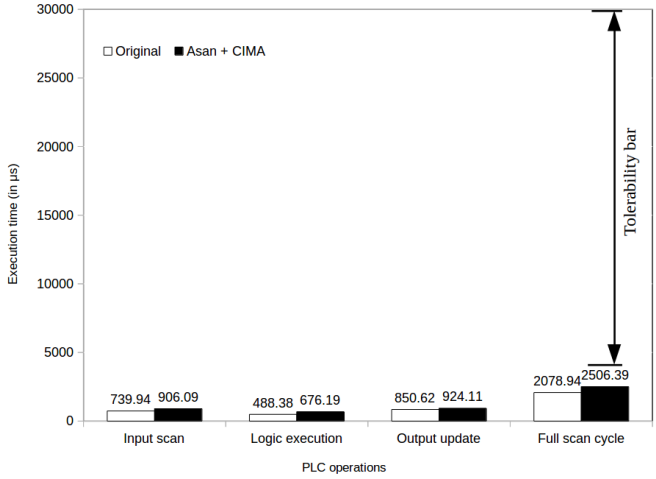


Figure 12: Tolerability of the worst-case overhead for Open-SecUTS testbed

Table 4: Memory usage overheads for the Open-SWaT Testbed

Category	Original	ASan		ASan+CIMA	
		Original	Overhead	Original	Overhead
Virtual Memory	62.97MB	549.38MB	8.72×	557.5MB	8.85×
Real Memory	8.17MB	10.31MB	1.26×	11.2MB	1.37×
Binary	144KB	316KB	2.19×	324KB	2.25×
Shared library	3196KB	4288KB	1.34×	4288KB	1.34×

6.3. Resilience

One of the main contributions of our work is to empirically show the resilience of our CIMA approach against memory-safety attacks. Here, we evaluate how our mitigation strategy ensures *availability* and *physical-state resiliency* of a real-world CPS. As discussed in the preceding sections, CIMA does not render system unavailability. This is because it does not abort or restart the PLC when mitigating memory-safety attacks. In such a fashion, the availability of PLCs is ensured by our ap-

Table 5: Memory usage overheads for the Open-SecUTS Testbed

Category	Original	ASan		ASan+CIMA	
		Original	Overhead	Original	Overhead
Virtual Memory	56.37MB	489.29MB	8.68×	490.6MB	8.70×
Real Memory	8.76MB	9.81MB	1.12×	10.21MB	1.17×
Binary	136KB	288KB	2.12×	296KB	2.18×
Shared library	3196KB	4288KB	1.34×	4288KB	1.34×

proach. As discussed in Section 3.2.2, physical-state resiliency of a CPS can be affected by the memory-safety overhead (when the overhead is not tolerable due to the real-time constraint of the PLC) or the downtime of the PLC (when the PLC is unavailable for some reason).

In the preceding section, we showed that the overall overhead is tolerable, i.e., $\hat{T}_s \leq T_c$. Hence, the additional overhead induced by ASan and CIMA does not affect the physical-state resiliency. Added to the fact is that the availability of SWaT and SecUTS is also ensured by CIMA via its very construction, as CIMA never aborts the system or leads to PLC downtime. That means, the downtime (i.e. τ) is zero, hence Eq. (8) is satisfied. This then ensures physical-state resiliency. This makes CIMA to be a security solution that ensures availability and maintains resilience of the CPS even in the presence of memory-safety attacks.

6.4. Memory usage overheads

Finally, we evaluated the memory usage overheads of our CIMA approach. Tables 4 and 5 summarize the increased virtual memory usage, real memory usage, binary size and shared library usage for the Open-SWaT and Open-SecUTS testbeds, respectively. The reported statistics are collected by reading `VmPeak`, `VmRSS`, `VmExe` and `VmLib` fields, respectively, from */proc/self/status*. In general, we observe a significant increase in virtual memory usages (8.85× for Open-SWaT and 8.70× for Open-SecUTS). This is primarily because of the allocation of large redzones with *malloc* (as part of the ASan approach). However, the real memory usage overhead is only 1.37× (for Open-SWaT) and 1.17× (for Open-SecUTS). We believe these overheads are still acceptable since most PLCs nowadays come with at least 1GB memory size. Moreover, the increased memory size is an acceptable trade-off in the light of strong mitigation mechanics provided by our CIMA approach. Finally, we observe that CIMA introduces negligible memory usage overhead over ASan, meaning the majority of memory-usage overhead is attributed to the usage of ASan.

7. Related work

CIMA is built on top of ASan [9]. The classic version of ASan covers a wide range of temporal memory errors, such as use-after-free, use-after-return and memory leaks, and spatial memory errors such as stack, heap and global buffer overflows. However, in contrast to CIMA, ASan does not provide any mitigation (from the point of view of availability) against memory safety attacks.

Widespread system-based countermeasures against memory-safety attacks, such as stack canaries [54, 55], address space layout randomization (ASLR) [56, 57, 58], position independent execution (PIE) [59], non-executable memory page (NX) [60], etc., are also not designed to guarantee availability but to prevent other exploits, and have also been challenged by recent attack techniques such as return oriented programming [61, 7].

Softbound [18] and its extension CETS [19] guarantee a complete memory-safety. However, such guarantees arrive with the cost of a very high runtime overhead (116%). Such a high overhead is unlikely to be tolerable for the real-time constraints imposed on CPS. Moreover, Softbound and CETS do not implement a mitigation strategy to consider the physical-state resiliency in a CPS context. A different work, named SafeDispatch [12] imposes much lower overhead (2.1%) in contrast to Softbound and CETS, yet does not provide any mitigation against memory-safety attacks.

Rinard et al. [13] developed "failure-oblivious computing", which allows a vulnerable program to continue execution even in the presence of memory errors. To achieve this, manufactured values were returned for invalid reads and invalid writes were simply discarded. However, this approach has several limitations. First, providing a fabricated value is not acceptable and might lead to serious consequences for CPS. For example, providing fabricated sensor inputs to the PLC may mislead the PLC to issue incorrect control commands which will eventually affect the CPS dynamics. Furthermore, these fabricated values may also determine branches or loop conditions and may lead the program to an unexpected state. Secondly, the "failure-oblivious computing" technique is designed only against buffer overflow vulnerabilities; it does not cover other critical memory-safety vulnerabilities. Finally, this technique is designed for mainstream systems, such as Servers, hence its applicability in the CPS environment was never validated.

Sting [25, 26] is an end-to-end self-healing architecture developed against memory-safety attacks. However, as its architecture is based on address space layout randomization (ASLR) and system-call-based anomaly detection, the provided defense can be bypassed via code-reuse attacks and data injection attacks. Moreover, Sting employs periodic check-pointing to resume the victim program from an earlier safe state. This, in turn, induces high performance and memory usage overhead [25], leading to unavailability of the system. As reported in [25], there are also cases where the corrupted program cannot be recovered and restarting the program is required. Because of these limitations, the applicability of Sting in a CPS environment is limited.

SafeDispatch [12] is a fast memory-safety tool developed within the LLVM infrastructure. SafeDispatch also involves exhaustive performance optimizations to make the overhead just 2.1%. However, SafeDispatch is not supported by an appropriate mitigation strategy that guarantees system availability in the presence of memory-safety attacks.

ROPocop [14] is a dynamic binary code instrumentation framework against code injection and code reuse attacks. It relies on Windows x86 binaries to detect such attacks, hence not applicable on Linux-based systems. It also introduces a high runtime overhead of 240%, which is unlikely be tolerable for the hard real-time constraints in CPS. Moreover, there is no denial of service mitigation strategy presented with this solution.

Over the past decades, a number of control-flow integrity (CFI) based solutions (e.g., [15, 16, 17]) have been developed to defend against memory-safety attacks. The main objective behind these solutions is ensuring the control-flow integrity of a program. Therefore, they aim to prevent attacks from redi-

recting the flow of program execution. These solutions also offer a slight performance advantage over other countermeasures, such as code-instrumentation based countermeasures. However, CFI-based solutions generally have the following limitations: *(i)* determining the required control-flow graph (often using static analysis) is difficult and requires a substantial amount of memory; *(ii)* attacks that are not diverting the control-flow of the program cannot be detected (e.g. data oriented attacks [62]); *(iii)* finally, these solutions are only to detect memory-safety attacks, but does not provide any mitigation strategy to ensure system availability.

ECFI [24] is a control-flow based strategy developed against control-flow hijacking attacks. It effectively detects memory-safety attacks with low overhead. However, it is a passive system and cannot detect memory-safety attacks proactively. Since it is also a CFI-based solution, it suffers from the limitations discussed in the preceding paragraph. Specifically, it only offers detection of memory-safety attacks. However, in contrast to our work, ECFI does not mitigate these attacks to improve availability of the system.

The introduction of just-In-Time (JIT) compilers is a hybrid interpretation/compilation technique enhancements in program execution time. However, it also brings security risks as JIT compilers rely on writable memory where it places dynamically generated code [63]. This principle of JIT conflicts/clashes against the already established (system-based) countermeasures against code injection attacks, such as non-executable memory (NX) [60] or data execution protection (DEP) countermeasures.

In order to coexists with such system-based countermeasures, programmers using JIT often need to bypass them by manually allowing execution of code in memory pages with write access for instance, opening the door to attacks. Therefore, memory protection mechanisms for JIT compilers [63], [64], [65] focus on mantaining a balance between security and JIT, with a focus on preventing code injection attacks. Our solution differs from those approaches in that we are focused on guaranteeing availability while preventing exploitation, and as such go beyond access control strategies in memory regions and have to deal with a (security motivated) modification of the control flow of the application when under attack.

In summary, to the best of our knowledge, there is no prior work that efficiently detects and mitigates memory-safety attacks without compromising the real-time constraints or availability of the underlying system. In this paper, CIMA provides a practical solution to bridge this gap in the research.

8. Conclusion

In this paper, we propose CIMA, a resilient mitigation strategy to ensure protection against a wide variety of memory-safety attacks. The main advantage of CIMA is that it mitigates memory-safety attacks in a time-critical environment and ensures the system availability by skipping only the instructions that exhibit illegal memory accesses. To this end, we evaluate our approach on a real-world CPS testbeds. Our evaluation reveals that CIMA mitigates memory-safety errors with acceptable runtime and memory-usage overheads. Moreover, it ensures that the resiliency of the physical states are maintained despite the presence of memory-safety attacks. Although we evaluated our approach only in the context of CPS, CIMA is also applicable and useful for any system under the threat of memory-safety attacks.

In future, we plan to build upon our approach to understand the value of CIMA across a variety of systems beyond CPS. We also plan to leverage our CIMA approach for live patching. In particular, at its current state, CIMA does not automatically fix the memory-safety vulnerabilities of the victim code (although CIMA does ensure that the vulnerabilities are not exploited by an attacker at runtime). Instead, it generates a report for the developer to help produce a patched version of the code. In future, we will use the report generated by CIMA to automatically identify the pattern of illegal memory accesses and fix the code accordingly. As a compile-time tool, the current state of CIMA is dependent on the availability of the source code to mitigate memory-safety attacks. Therefore, binary instrumentation with CIMA is left as a future work.

Acknowledgment

This work was supported by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2014NCR-NCR001-31) and administered by the National Cybersecurity R&D Directorate. This work was also partially supported by the Ministry of Education of Singapore under the grant MOE2018-T2-1-098

References

[1] L. Szekeres, M. Payer, T. Wei, D. Song, Sok: Eternal war in memory, IEEE Symposium on Security and Privacy (2013).

[2] T. Saito, R. Watanabe, S. Kondo, S. Sugawara, M. Yokoyama, A survey of prevention/mitigation against memory corruption attacks, in: International Conference on Network-Based Information Systems (NBiS), 2016.

[3] Y. Younan, W. Joosen, F. Piessens, Code injection in c and c++ : A survey of vulnerabilities and countermeasures, Tech. rep. (2004).

[4] A. Francillon, C. Castelluccia, Code injection attacks on harvard-architecture devices, in: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08), 2008. doi:10.1145/1455770.1455775.

[5] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, A. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: Proceedings of the IEEE Symposium on Security and Privacy (SP'13), Washington, USA, 2013. doi:10.1109/SP.2013.45.

[6] J. Dahse, N. Krein, T. Holz, Code reuse attacks in php: Automated pop chain generation, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'14), 2014.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Maziéres, D. Boneh, Hacking blind, in: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP'14, 2014.

[8] E. G. Chekole, J. H. Castellanos, M. Ochoa, D. K. Y. Yau, Enforcing memory safety in cyber-physical systems, in: Katsikas S. et al. (eds) Computer Security. SECPRE 2017, CyberICPS 2017, 2017.

[9] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, Addresssanitizer: a fast address sanity checker, in: Proceedings of the USENIX conference on Annual Technical Conference (USENIX ATC'12), 2012.

[10] E. G. Chekole, S. Chattopadhyay, M. Ochoa, G. Huaqun, Enforcing full-stack memory safety in cyber-physical systems, in: Proceedings of the

International Symposium on Engineering Secure Software and Systems (ESSoS'18), 2018.
[11] L. Hu, N. Xie, Z. Kuang, K. Zhao, Review of cyber-physical system architecture, in: IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2012.
[12] D. Jang, Z. Tatlock, S. Lerner, Safedispatch: Securing c virtual calls from memory corruption attacks, Proceedings 2014 Network and Distributed System Security Symposium (2014).
[13] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, W. S. Beebee, Jr., Enhancing server availability and security through failure-oblivious computing, in: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, 2004.
[14] A. Follner, E. Bodden, Ropocop — dynamic mitigation of code-reuse attacks, Journal of Information Security and Applications (2016).
[15] M. Zhang, R. Sekar, Control flow integrity for cots binaries, in: Proceedings of the USENIX Security Symposium (USENIX'13), 2013.
[16] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, G. Pike, Enforcing forward-edge control-flow integrity in gcc & llvm, in: Proceedings of the 23rd USENIX Security Symposium, USENIX'14, 2014, pp. 941–955.
[17] X. Ge, N. Talele, M. Payer, T. Jaeger, Fine-grained control-flow integrity for kernel software, in: 2016 IEEE European Symposium on Security and Privacy, 2016. doi:10.1109/eurosp.2016.24.
[18] S. Nagarakatte, J. Zhao, M. M. Martin, S. Zdancewic, Softbound: Highly compatible and complete spatial memory safety for C, in: Proceedings of the SIGPLAN conference on Programming language design and implementation, 2009.
[19] S. Nagarakatte, J. Zhao, M. M. Martin, S. Zdancewic, Cets: Compiler enforced temporal safety for C, in: Proceedings of the 2010 International Symposium on Memory Management (ISMM'10), 2010.
[20] M. S. Simpson, R. K. Barua, Memsafe: Ensuring the spatial and temporal memory safety of c at runtime, Software: Practice and Experience 43 (1) (2013).
[21] D. Bruening, Q. Zhao, Practical memory checking with dr. memory, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2011.
[22] G. C. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer, Ccured: Type-safe retrofitting of legacy software, ACM Trans. Program. Lang. Syst. (2005).
[23] F. Ch. Eigler, Mudflap: pointer use checking for C/C++, in: GCC Developer's Summit, 2003.
[24] A. Abbasi, T. Holz, E. Zambon, S. Etalle, Ecfi: Asynchronous control flow integrity for programmable logic controllers, in: Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC 2017, 2017, pp. 437–448.
[25] J. Newsome, D. Brumley, D. Song, Sting: An end-to-end self-healing system for defending against zero-day worm attacks on commodity software (2005).
[26] J. Newsome, D. Brumley, D. Song, Vulnerability-specific execution filtering for exploit prevention on commodity software, in: Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS'05), 2005.
[27] D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha, Towards automatic generation of vulnerability-based signatures, in: Proceedings of IEEE Symposium on Security and Privacy (SP'06), DC, USA, 2006. doi:10.1109/SP.2006.41.
[28] A. Smirnov, T. Chiueh, Automatic patch generation for buffer overflow attacks, in: International Symposium on Information Assurance and Security, 2007.
[29] CVE-2016-5814, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5814 (2016).
[30] CVE-2012-6438, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6438 (2012).
[31] CVE-2012-6436, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6436 (2012).
[32] CVE-2013-0674, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0674 (2013).
[33] CVE-2015-1449, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1449 (2015).
[34] CVE-2012-0929, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0929 (2012).
[35] CVE-2015-7937, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7937 (2015).
[36] CVE-2011-5007, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5007 (2011).
[37] S. Adepu, A. Mathur, Using process invariants to detect cyber attacks on a water treatment system, in: ICT Systems Security and Privacy Protection, 2016.
[38] C. M. Ahmed, C. Murguia, J. Ruths, Model-based attack detection scheme for smart water distribution networks, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17, 2017, pp. 101–113.
[39] C. M. Ahmed, M. Ochoa, J. Zhou, A. P. Mathur, R. Qadeer, C. Murguia, J. Ruths, Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems, in: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, 2018, pp. 483–497.
[40] C. Murguia, J. Ruths, Characterization of a cusum model-based sensor attack detector, in: IEEE 55th Conference on Decision and Control (CDC), 2016, pp. 1303–1309.
[41] A. github repository, Comparison of addresssanitizer with other memory safety tools, https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools (2015).
[42] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, T. Holz, Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c applications, 2015 IEEE Symposium on Security and Privacy (2015). doi:10.1109/sp.2015.51.
[43] E. G. Chekole, G. Huaqun, Ics-sea: Formally modeling the conflicting design constraints in ics, in: Proceedings of the Fifth Annual Industrial Control System Security (ICSS) Workshop, ICSS, Association for Computing Machinery, New York, NY, USA, 2019, p. 60–69. doi:10.1145/3372318.3372325.
URL https://doi.org/10.1145/3372318.3372325
[44] GCC, Gimple, https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html (2018).
[45] GCC, Gcc basic blocks, https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html (2018).
[46] Common Vulnerabilities and Exposure (CVE), Cve-2018-20818, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20818 (2019).
[47] E. G. Chekole, U. Cheramangalath, S. Chattopadhyay, M. Ochoa, H. Guo, Taming the war in memory: A resilient mitigation strategy against memory safety attacks in CPS, CoRR abs/1809.07477 (2018). arXiv:1809.07477.
[48] G. Liang, J. Zhao, F. Luo, S. R. Weller, Z. Y. Dong, A review of false data injection attacks against modern power systems, IEEE Transactions on Smart Grid 8 (4) (2017) 1630–1638.
[49] SWaT, Secure water treatment (swat) testbed (2018).
[50] C. M. Ahmed, J. Zhou, A. P. Mathur, Noise matters: Using sensor and process noise fingerprint to detect stealthy cyber attacks and authenticate sensors in cps, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC'18, 2018, pp. 566–581.
[51] OpenPLC, Openplc, http://www.openplcproject.com/ (2018).
[52] ScadaBR, Scadabr, http://www.scadabr.com.br/ (2018).
[53] L. Zhou, H. Guo, D. Li, J. W. Wong, J. Zhou, Mind the gap: Security analysis of metro platform screen door system, in: Proceedings of the Singapore Cyber-Security RandD Conference (SG-CRC'17), 2017.
[54] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, in: Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, USENIX Association, 1998.
[55] Duartes.org, Epilogues, canaries, and buffer overflows, http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/.
[56] LWN.net, Address space layout randomization, https://lwn.net/Articles/569635/.
[57] S. Bhatkar, D. C. DuVarney, R. Sekar, Address obfuscation: An efficient approach to combat a board range of memory error exploits, in: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, USENIX Association, 2003.
[58] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, in: Proceedings of

the 11th ACM Conference on Computer and Communications Security, CCS'04, 2004.

[59] fpmurphy.com, Position independent executables, http://blog.fpmurphy.com/2008/06/position-independent-executables.html.

[60] Solar Designer, Non-executable user stack, https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/full_papers/cowan/cowan_html/node21.html.

[61] R. Roemer, E. Buchanan, H. Shacham, S. Savage, Return-oriented programming: Systems, languages, and applications, ACM Transactions on Information and System Security 15 (1) (2012).

[62] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, Z. Liang, Data-oriented programming: On the expressiveness of non-control data attacks, 2016 IEEE Symposium on Security and Privacy (2016).

[63] W. D. Groef, N. Nikiforakis, Y. Younan, F. Piessens, Jitsec: Just-in-time security for code injection attacks, in: Benelux Workshop on Information and System Security (WISSEC), 2010.

[64] M. Jauernig, M. Neugschwandtner, C. Platzer, P. M. Comparetti, Lobotomy: An architecture for jit spraying mitigation, in: Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security, ARES'14, IEEE Computer Society, USA, 2014, p. 50–58. doi:10.1109/ARES.2014.14.
URL https://doi.org/10.1109/ARES.2014.14

[65] B. Niu, G. Tan, Rockjit: Securing just-in-time compilation using modular control-flow integrity, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'14), 2014.