SCOPE: Secure Compiling of PLCs in Cyber-Physical Systems

Eyasu Getahun Chekole[1,*], Martín Ochoa[2,3], and Sudipta Chattopadhyay[3]

[1] Institute for Infocomm Research, A*STAR, Singapore 138632, Singapore
[2] Singapore University of Technology and Design, Singapore 487372, Singapore
[3] AppGate Inc, Bogotá, Colombia

Abstract. Cyber-Physical Systems (CPS) are being widely adopted in critical infrastructures, such as smart grids, nuclear plants, water systems, transportation systems, manufacturing, and healthcare services, among others. However, the increasing prevalence of cyberattacks targeting them raises a growing security concern in the domain. In particular, memory-safety attacks, that exploit memory-safety vulnerabilities, constitute a major attack vector against real-time control devices in CPS. Traditional IT countermeasures against such attacks have limitations when applied to the CPS context: they typically incur in high runtime overheads; which conflicts with real-time constraints in CPS and they often abort the program when an attack is detected, thus harming availability of the system, which in turn can potentially result in damage to the physical world. In this work, we propose to enforce a *full-stack* memory-safety (covering *user-space* and *kernel-space* attack surfaces) based on secure compiling of PLCs to detect memory-safety attacks in CPS. Furthermore, to ensure availability, we enforce a resilient mitigation technique that bypasses illegal memory access instructions at runtime by dynamically instrumenting low-level code. We empirically measure the computational overhead caused by our approach on two experimental settings based on real CPS. The experimental results show that our approach effectively and efficiently detects and mitigates memory-safety attacks in realistic CPS.

Keywords: Critical Infrastructures Security · CPS Security · Software Security · Memory Safety · Efficiency · Resilience · Availability

1 Introduction

Cyber-physical systems [48, 31, 32] are being widely adopted in various mission critical infrastructures including smart grids, water treatment and distribution systems, transportation, nuclear plants, robotics and manufacturing, among others. Despite their importance in such critical infrastructures, the increasing cyberattacks targeting them poses a growing security concern. One important class of cyberattacks in CPS are memory-safety attacks [52, 44] that target programmable logic controllers (PLCs).

A typical PLC consists of three main software components – the PLC firmware, the control software (i.e. the control logic) and the underlying OS hosting the PLC. Since these software components are commonly implemented in C/C++ languages (for the sake of efficiency), they are susceptible to memory-safety vulnerabilities, such as buffer overflows, use-after-free errors (dangling pointers), use-after-return errors, initialization order bugs, and memory leaks. Consequently, a wide-range of these vulnerabilities are being regularly discovered even in modern PLCs [20, 16, 15, 17, 18, 14, 19, 13] and Linux kernels [38].

These vulnerabilities could lead to runtime crashes, which can severely affect safety- and availability-critical systems, such as CPS. More importantly, these vulnerabilities can also be exploited by memory-safety attacks. Memory-safety attacks, such as code-injection [24] and code-reuse [51] attacks, can corrupt the memory system of a vulnerable program to hijack or subvert its operations. In CPS, these attacks

[*] *This work was partly done when **E.G. Chekole** was a Ph.D. student at the Singapore University of Technology and Design.*

can target the PLC's firmware and control software (*user-space*) or the underlying OS kernel hosting the PLC (*kernel-space*). Therefore, both the runtime crashes and memory-safety attacks are critical concerns in CPS.

To overcome the runtime crashes and security challenges, a wide-range of countermeasures, often referred as *memory-safety tools*, have been developed [47, 30, 34, 35, 36, 22, 50, 23, 6, 21, 29, 1, 54, 53, 26, 4, 37, 12, 41, 25, 43, 28]. However, the hard real-time and availability requirements imposed in CPS, alongside the use of resource-constrained edge devices, limit the practical applicability of certain memory-safety tools available. This is because, the high memory-safety overheads (MSO) induced by certain memory-safety tools compromise the real-time requirements in CPS. Furthermore, the non-resilient mitigation strategies exerted in certain memory-safety tools (e.g. plainly aborting/restarting the victim system when a memory-safety attack is detected) compromise availability of the system. Therefore, the *efficiency* and *mitigation resilience* of memory-safety tools are crucial requirements in CPS that should be met alongside strong security guarantees.
Efficiency – Most memory-safety tools, especially the code-instrumentation ones incur in high runtime overheads, e.g., RopoCop [23] (240%), CUP [6] (158%), CCured [36] (150%), SoftBoundCETS [34] (116%), and MemSafe [50] (87%). This high overhead may unacceptably compromise performance of the CPS. If the CPS real-time constraints are not met, major consequences can follow such as disruption of the control-loop stability, incorrect control by the use of stale information, availability issues, system damage (in the worst case), etc. Thus, the trade-off between security and efficiency remains as one of the main conflicting design challenges in CPS.
Mitigation resilience – Most of the existing memory-safety tools do not have a resilient mitigation strategy. They are primarily designed to abort or reboot the victim system when a memory-safety attack or violation is detected, thus leading to system unavailability. Such ineffective mitigation strategies are not acceptable in systems with stringent availability requirements, such as CPS. Because, system unavailability in CPS leaves the control system into an unsafe state and leads disruption of the CPS dynamics, which may result in a complete system failure (cf. Section 2.2). Thus, system availability is also a critical requirement in CPS.

Therefore, vis-a-vis the real-time and availability requirements (which we particularly associate with the efficiency and mitigation resilience of the security solutions) are equally critical as the runtime crashes and security concerns in the CPS environment.

Our approach To address these challenges, we propose a countermeasure called **S**ecure **C**ompiling **O**f **P**LCs in cyb**E**r-physical systems (**SCOPE**). Inspired by our recent work, CIMA [9], our approach is based on the intuition of proactively stopping memory-safety violations from happening, thereby preventing both runtime crashes and memory-safety attacks in the process. To accomplish this, we follow a compile-time code-instrumentation based approach that offers stronger guarantees in terms of error coverage and detection accuracy, despite introducing higher performance overheads. To cover the attack surfaces in user-space and kernel-space, we escalate our solution to a full-stack memory-safety countermeasure, comprising a user- and kernel-space memory-safety solutions.

After researching over several available tools, we port the popular memory-safety tools, such as AddressSanitizer(ASan) [47] and Kernel Address Sanitizer (KASan) [30], as a user-space and kernel-space memory error detector tools, respectively, by fixing their limitations to work in a CPS environment. We enhance this detection strategy by integrating our recent mitigation work, CIMA[9], that systematically combines a compile-time code-instrumentation and runtime monitoring techniques to resiliently mitigate the detected memory-safety attacks.

This work is an extension of our previous works. In brief, it combines our prior memory-safety works on: 1) *a user-space attack detection* [7] (disregarding availability attacks (i.e. no mitigation resilience) and without considering a kernel-space memory-safety); 2) *a user-space mitigation resilience* [9] (without considering a kernel-space memory-safety); and 3) *a kernel- and user-space attack detection* [8] (disregarding availability attacks). So, in this work, we integrated the three approaches together to form a full-stack memory safety. Although we previously studied them separately, ultimately they should all together be part of the secure compilation strategy. It is an open question whether a practical CPS would tolerate the joint computational overhead induced by the proposed full-stack memory-safety.

Evaluation The effectiveness of the proposed full-stack memory-safety is experimentally evaluated. Our experiments are based on two realistic CPS testbeds: SWaT (Secure Water Treatment System) [33] and SecUTS (Secure Urban Transportation System) [55], comprising real-world vendor-supplied PLCs. However, the vendor's PLC firmware (both in SWaT and SecUTS) is closed-source, hence we could not incorporate our memory-safety solutions in these PLCs. To circumvent this challenge, we prototyped our experimental testbeds, which we call open-SWaT and Open-SecUTS, using open-source PLCs to mimic the behavior of SWaT and SecUTS, respectively, according to their detailed operational profiles. Then, we report experiments conducted on Open-SWaT and Open-SecUTS.

A strong memory-safety countermeasure apparently incurs high cost, i.e., performance overhead, which might not be acceptable in CPS due to the hard real-time and availability requirements imposed in these systems. To evaluate the acceptability of such overheads, we briefly modeled the CPS design constraints, such as the *real-time* and *physical-state resiliency* requirements. These models aid as benchmarks to evaluate tolerability of the performance overheads and resilience of the system dynamics in CPS. Subsequently, we evaluated the effectiveness of our full-stack memory-safety solution on the Open-SWaT and Open-SecUTS testbeds. In particular, we evaluate tolerability of the induced performance overhead in accordance with the CPS real-time constraints we modeled. Our experimental results on Open-SWaT and Open-SecUTS reveal that the introduced memory-safety overhead of 91.02% (for Open-SWaT) and 85.49% (for Open-SecUTS) would not impact the normal operations of SWaT and SecUTS. Furthermore, our user-space mitigation strategy also meets physical-state resiliency of the CPS testbeds under test.

In general, our full-stack countermeasure efficiently and successfully prevents memory-safety violations from happening without compromising availability of the system. To the best of our knowledge, this is not achieved by any prior work. Although our proposed memory-safety is applicable for any computing system involving C/C++ programs, we particularly focused on the CPS domain in this research. This is because, unlike the mainstream systems, CPS often imposes conflicting design constraints including real-time guarantees and physical-state resiliency – involving its physical dynamics and security. Note that attacks that manipulate sensor or actuator values at storage or communication levels are out of our scope, and can be handled via orthogonal approaches, e.g., using physics-based approaches [27], machine-learning techniques [39], or access-control mechanisms [3, 2].

In sum, the proposed work tackles the problem of *quantifying the practical tolerability of enforcing a strong full-stack memory-safety on realistic CPS with hard real-time constraints and limited computational power.* Furthermore, *this work tackles the problem of ensuring availability of critical services and systems while successfully detecting and mitigating a wide-range of memory-safety attacks.*

We make the following contributions: **a)** The enforced full-stack memory-safety effectively prevents both runtime crashes (that could arise due to memory-safety violations) and memory-safety attacks in CPS, both in user-space and kernel-spaces. **b)** We formally define and model the notions of real-time and physical-state resiliency constraints, that are crucial in the context of CPS. **c)** We empirically measure and quantify tolerability of the induced performance overhead of our full-stack memory-safety based on the real-time constraints of two realistic CPS systems. **d)** Our user-space memory-safety ensures system availability and physical-state resiliency with reasonable performance and storage overheads. Therefore, it is practically applicable to systems with stringent timing constraints, such as CPS, beyond the mainstream systems. **e)** The efficiency and effectiveness of our approach is evaluated on two real-world CPS testbeds containing vendor-supplied PLCs.

2 Attacker and system models

This section discusses our attacker model and the CPS design constraints we formally modelled.

2.1 Attacker model

The main objective of memory-safety attacks (e.g. code-injection and code-reuse attacks) is to get a privileged access or to take control of the vulnerable system. To achieve this, the attacker exploits memory-safety vulnerabilities, e.g., buffer over/under-flows and dangling pointers, that can be found in the targeted program. We briefly illustrate the exploitation strategy using a simple C/C++ program consisting of a buffer overflow vulnerability (cf. Program 1). A relevant memory layout of the program is provided in Figure 1a, just to simplify the illustration of the exploitation strategy. This includes the defined buffer address and extended instruction pointer (EIP) of the program.

The vulnerable function, i.e., *"gets(buffer)"*, allows the attacker to send an input data that is larger than the allocated buffer size. The attacker can exploit this vulnerability by creating a systematically tailored input that serves to overwrite the buffer's boundary, the EIP and other important memory addresses. In brief, the tailored input consists of the *attacker defined memory address* (e.g. 0×xy in Figure 1b) – which will serve for overwriting the EIP, and a *malicious code* – to be injected into the program's address space (in case of code-injection attacks). Figure 1b illustrates the tailored input. As shown in Figure 1c, the attacker defined address is made to point the starting address of the injected malicious code (for code-injection attacks) or existing system modules (for code-reuse attacks). The attack will be then launched by sending the tailored input to the buffer. The exploitation strategies are briefly illustrated in Figure 2. A detailed account of such exploitation strategies can also be found in [5, 45].

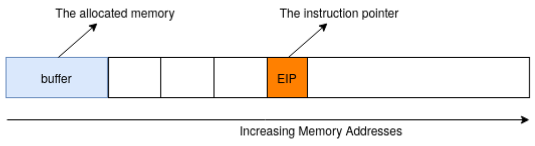
Program 1: A code snippet containing a simple buffer overflow vulnerability.

```
foo() {
    char buffer[16];
    printf("Insert input: ");
    gets(buffer);
}
```

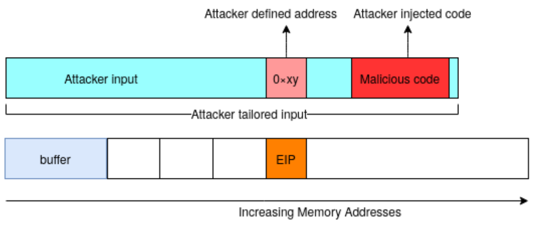
2.2 Modeling CPS design constraints

Unlike traditional IT systems, CPS involves complex and continuous interactions between entities in the physical and cyber spaces over communication networks. These interactions are accomplished via communications with the physical-world through sensors and actuators and with the digital-world through PLCs (controllers) and other embedded devices. An abstraction of a typical CPS is illustrated in Figure 3.

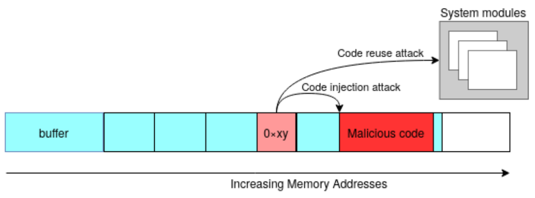
The interactions among CPS entities, such as sensors, PLCs and actuators, is synchronized via system time. These interactions, unlike in conventional IT systems, are constrained by hard deadlines. Missing deadlines could result in disruption of the control-loop stability or damage to the physical plant (in the worst case). Because, such situations could lead the underlying system to run into an unsafe and unstable states. This is the main reason that makes CPS to be highly delay sensitive and real-time constrained systems.



(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)

Fig. 1: A high-level illustration of memory-safety attacks exploitation strategies.

In particular, since PLCs form the main control devices in CPS, the real-time requirements are particularly imposed in these devices to maintain the safety and stability of the control system in CPS. To be able to formally capture the timing constraints in CPS, we define two crucial notions, namely *real-time constraints* and *physical-state resiliency*, which will serve as metrics to evaluate the efficiency and resilience of our full-stack memory-safety enforcement, respectively. We formally define and discuss these notions in the following sections.

Real-time constraints As shown in Figure 4, PLCs undergo a continuous and cyclic process when issuing control commands to actuators. This process involves three main operations, namely input scan, PLC logic execution and output update. This cyclic process is often referred as the PLC's *scan cycle*. The overall time it takes to complete the scan cycle (i.e. to execute the three operations) is referred as the *scan time* (T_s) of the PLC [11]. To effectively synchronize the interactions and communications among its various entities, a typical CPS defines an upper-bound scan time to each PLC, called *cycle time* (T_c). Meaning, each scan cycle has to be completed within the specified cycle time of the PLC, i.e., $T_s \leq T_c$. We define this requirement as the *real-time constraint* of the PLC. A typical PLC meets this constraint by design. However, due to security overheads, such as MSO, PLCs might not meet this constraint. For example, by hardening the PLC with our memory-safety protection, the scan time increases. This increase in the scan time is attributed to the MSO. Concretely, the MSO can be computed as follows:

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

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

The induced MSO by the memory-safe compilation obviously causes a delay on the PLC operations. However, it is essential to check whether this MSO 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 brief, after securely compiling the PLC with our memory-safety, we measure the PLC scan time, i.e. $\hat{T}_s$, for n different scan cycles. Then, we compute the MSO in average-case (i.e. $mean(\hat{T}_s)$) and worst-case

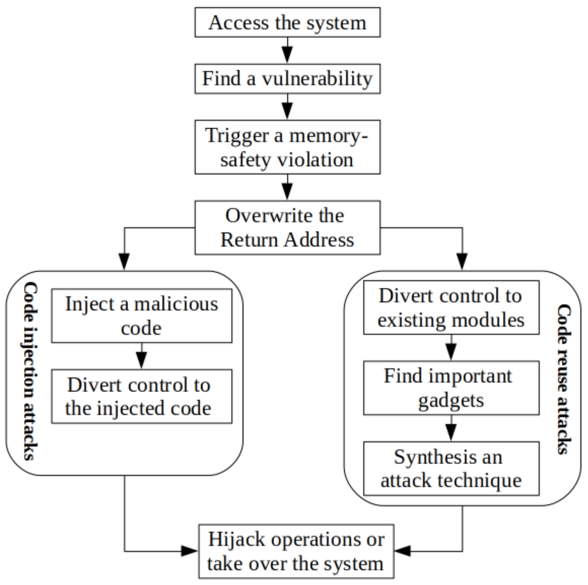

Access the system
Find a vulnerability
Trigger a memory-safety violation
Overwrite the Return Address
Code injection attacks
Inject a malicious code
Divert control to the injected code
Code reuse attacks
Divert control to existing modules
Find important gadgets
Synthesis an attack technique
Hijack operations or take over the system

Fig. 2: Memory-safety exploitation strategies.

(i.e. $max(\hat{T}_s)$) scenarios. 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.

Similarly, the MSO is acceptable in the worst-case if the following condition is satisfied:

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

Note that since cyber-physical systems are hard real-time constrained systems, each scan time should meet the PLC's real-time requirement. As such, the worst-case scenario should be used to ensure real-time guarantees in CPS. However, the worst-case MSO (i.e. the highest cycle time obtained out of n

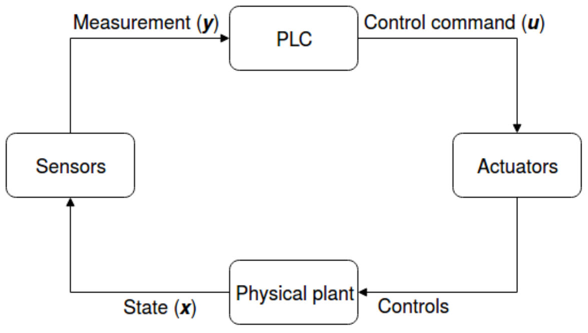


Fig. 3: An abstraction of a typical CPS
[**Acronyms:** x = state vector, y = sensor measurements, u = control command]

scan cycles) might not reflect the actual overhead of the enforced memory-safety. This is because, some scan times could be inflated even due to non-MSO related reasons, e.g. sudden execution interruptions due to unforeseen reasons. For this reason, it is essential to demonstrate the average MSO as well since it gives an intuition of the average performance penalties to be paid when enforcing this memory-safety solution even in other CPS testbeds. Therefore, we demonstrate both the average-case (computed out of 50,000 scan cycles) and worst-case MSO in this paper.

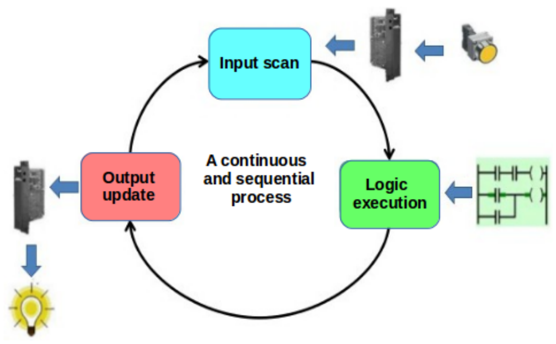


Fig. 4: The scan cycle of a PLC

Physical-state resiliency As discussed in the preceding sections, the stability of PLCs is crucial in enforcing the dynamics of a CPS to be compliant with its requirements. The PLCs real-time constraints, discussed in Section 2.2, play a crucial role to stabilize the PLC operations as well as to properly synchronize the interactions among various entities in CPS. However, these constraints are mainly for the cyber-world (where controls and communications take place) and might not clearly reflect the impact of control delays on the physical-world (where physical processes take place). In this section, we particularly model the impact of control delays (or PLC downtime, in other words) on the physical plant in CPS. For example, a PLC issues a control command at the rate of its cycle time (i.e. T_c) and the respective actuator also receives this command at the same rate. If a control delay happens for an arbitrary amount of time, say τ, then the actuator will not receive a fresh control command for the duration of τ. Consequently, the physical dynamics of the CPS will be affected for a total of $\frac{\tau}{T_c}$ scan cycles.

We note that the duration τ might be arbitrarily large depending on the reason that causes the control delay. For example, the control delay could happen because of the MSO (cf. Section 2.2) or as a result of a non-resilient mitigation strategy that typically aborts/restarts the PLC (i.e. causing PLC downtime) when a memory-safety attack is detected (cf. Section 3.1).

In either case, the scan time of the PLC with the enforced memory-safety (i.e. $\hat{T}_s$) may increase beyond the cycle time (i.e. T_c). This may affect the dynamics of the physical processes in CPS. In the worst case, this delay could cause damage to the CPS by violating its upper-bound or lower-bound physical state (i.e. x) limits of the plant.

For example, let us take the first process in SWaT (discussed in Section 4.1). 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 overflows when the water level surpasses the upper-bound limit of the tank. This occurs due to the control delay τ on PLC1, during which, the control command (i.e. "ON") computed by PLC1 may not change. Similarly, if the actuator receives the "OFF" command from PLC1 for an arbitrarily long duration, then the water tank underflows when the water level goes beyond the lower-bound limit of the tank. This is because tanks from other processes expect raw water from this underflow tank. The occurrence of such phenomena could severely affect the system dynamics in CPS.

Here, we quantify the *tolerability* of the control delay τ, i.e., the length of τ that does not violate the upper-bound and lower-bound physical state limits of the plant. We define this notion of tolerance as *physical-state resiliency.*

The tolerability of τ, in fact, depends on the current physical-state of the plant (e.g. water level, in case of PLC1 in SWaT) and the last control command issued by the PLC (e.g. "ON" or "OFF" command). In the following, we will formally define τ and the notion of *physical-state resiliency* in CPS.

To accurately formulate the control delay τ, we need to consider the following three mutually exclusive scenarios:

1. The PLC is aborted or restarted.
2. The PLC 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 CPS. This is because the PLCs, despite having increased scan time, still meet the real-time constraint T_c. Thus, they are not susceptible to control delays.
3. The PLC is neither aborted nor restarted and $\hat{T}_s > T_c$. In this case, the PLCs will have a control delay of $\hat{T}_s - T_c$, as the scan time violates the real-time constraint T_c.

Based on the intuitions discussed in the preceding paragraphs, we formally define τ as follows:

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

where $\mathit{\Delta}$ captures a non-deterministic threshold on the control delays when the PLC is aborted or restarted.

To formally model the physical-state resiliency, we will take a control-theoretic approach. For the sake of simplicity, 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 3):

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

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

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 control delay. With the control delay 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 there was a control delay 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 control delay τ. In Eq. (7), we assume, without loss of generality, that u_{t-1} is the last control command received from the PLC before the occurrence of the control delay.

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 $\omega \in \mathbb{R}^k$ and lower-bound $\theta \in \mathbb{R}^k$ thresholds on the physical state vector x_t. Therefore, to satisfy the physical-state resiliency, x_t must not exceed ω nor subceed θ. Formally, we say that a typical CPS (cf. Figure 3) satisfies physical-state resiliency if and only if the following condition holds at an arbitrary time index t:

$$\theta \le x'_{t+1} \le \omega$$
$$\theta \le Ax_t + Bu_{t-1}[\![t, t+\tau]\!] \le \omega \tag{8}$$

Figure 5 illustrates three representative scenarios to show the consequence of Eq. (8). If the control delay $\tau_1 = 0$, then u_t (i.e. control command at time t) is correctly computed and $x'_{t+1} = x_{t+1}$. If the control delay $\tau_2 \in (1, 2]$, 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 the control delay $\tau_3 > 2$, the control command vector u_{t+i} for $i \ge 0$ will be the same as u_{t-1}. As a result, the estimated state vectors x'_{t+j} for $j \ge 1$ will unlikely to be identical to x_{t+j}.

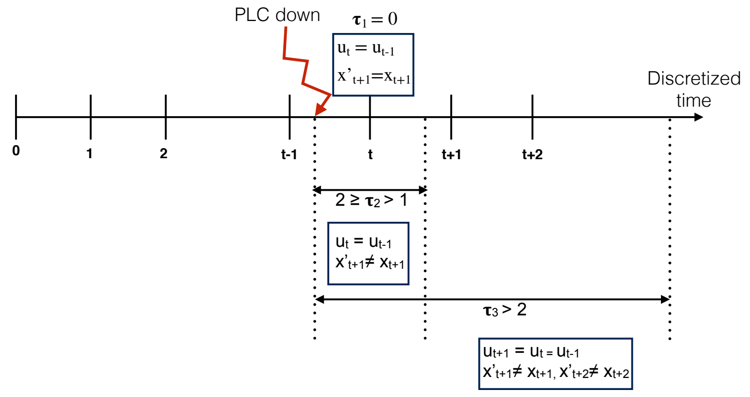


Fig. 5: Illustrating the impact of control delays in CPS.

3 Secure compiling of PLCs

In this section, we present a high-level discussion of the memory-safety tools we used in our full-stack memory-safety enforcement.

3.1 ASan: user-space detection

Overview Although several memory-safety tools are available, they might not be practically applicable in CPS because of various reasons. This includes limitations in error coverage, low detection accuracy, high performance overheads, ineffective mitigation strategy, and architectural incompatibilities, among others. After researching over various tools, we chose ASan [47] as our user-space memory error and attack detector tool because of its high detection accuracy, broader error coverage and relatively low performance overhead when compared with other code-instrumentation based memory-safety tools [47, 42].

ASan instruments C/C++ programs at compile-time and creates poisoned memory regions, known as *redzones*. These redzones are not addressable and any instruction attempting to access them will be proactively detected as a memory safety violation. The instrumentation of ASan is illustrated in

Figure 6 and 7. Such an instrumented code can detect numerous memory-safety vulnerabilities, such as buffer overflows (i.e. stack, heap and global buffer overflows), use-after-free errors (dangling pointers), use-after-return errors, initialization order bugs, memory leaks, and double free errors.

Porting ASan to CPS Since ASan is specifically designed for the x86 target architectures, it is incompatible with AVR or ARM based architectures in the CPS environment. In this work, we adapt ASan for ARM based PLC systems and evaluate its applicability in the context of cyber-physical systems. PLC instructions as well as firmware of the PLC are compiled and instrumented by a modified ASan to guarantee a notion of memory-safety when the program is executed at runtime.

Validation The main limitation of ASan is its non-resilient mitigation strategy; it simply aborts the victim program whenever a memory violation or an attack is detected (cf. Figure 6 and 7). Ultimately, cyber-physical systems require a notion of security that guarantees correct functioning of the control system under strong attacker models. However, ASan fails at that as the detection of a memory-safety violation leads to abortion of the PLC program and may leave the control system in an unsafe state. This makes ASan inapplicable in systems with stringent availability constraints, such as CPS.

To address this mitigation limitation, we recently implemented CIMA [9] – a resilient mitigation technique against memory-safety attacks – and integrated it with ASan (see Section 3.2). Thus, CIMA enhances the capability of ASan to mitigate memory-safety bugs on-the-fly.

3.2 CIMA: user-space mitigation

Overview CIMA [9] is a resilient mitigation strategy we recently implemented to address the mitigation limitation of ASan. Instead of aborting the victim program upon detection of a memory-safety violation, CIMA proactively counters memory-safety violations from occurring. This is accomplished by proactively skipping (i.e. not executing) the illegal memory access instructions , i.e., instructions that attempt to access memory illegally, at runtime. In such a way, CIMA effectively prevents memory-safety violation from occurring without aborting/restarting the victim system. This preserves system availability and also maintains physical-state resiliency in CPS even in the presence of memory-safety attacks.

Detailed methodology To bypass illegal memory accesses, CIMA systematically constructs and manipulates the compiler-generated control-flow graph (CFG) of the program. CIMA constructs the CFG at compile-time by instrumenting each memory access instruction of the program. The instrumentation involves computation of the target instruction T_i for each respective memory access instruction i. The target instruction T_i is computed as a single successor of the memory access instruction i in the CFG, which will be determined at runtime. So, the newly constructed CFG will contain each memory access instruction i and its corresponding target instruction T_i (cf. Figure 6 and 7). In such a fashion, if instruction i is detected as illegal at runtime, i will be then bypassed and its target instruction T_i is executed instead, hence preventing the execution of the illegal memory access instruction. If T_i is also detected as illegal, the successor of T_i will be then executed, and so on. The rest of the execution, nevertheless, continues without interruption. This ensures availability of the program even in the presence of memory-safety attacks or violations.

Furthermore, it is noteworthy that the construction of the new CFG involves two scenarios depending on the location of i and T_i in the original CFG.
Scenario 1: When the memory access instruction i and its target instruction T_i are resided in the same basic-block (say bb). In this case, it is not possible to make a conditional jump to T_i within the same basic-block bb if i is detected as illegal. Thus, to make the conditional jump possible, the basic block bb is split to two basic blocks – i_{bb} (containing i) and T_{bb} (containing T_i). Now, control can jump to T_{bb} if

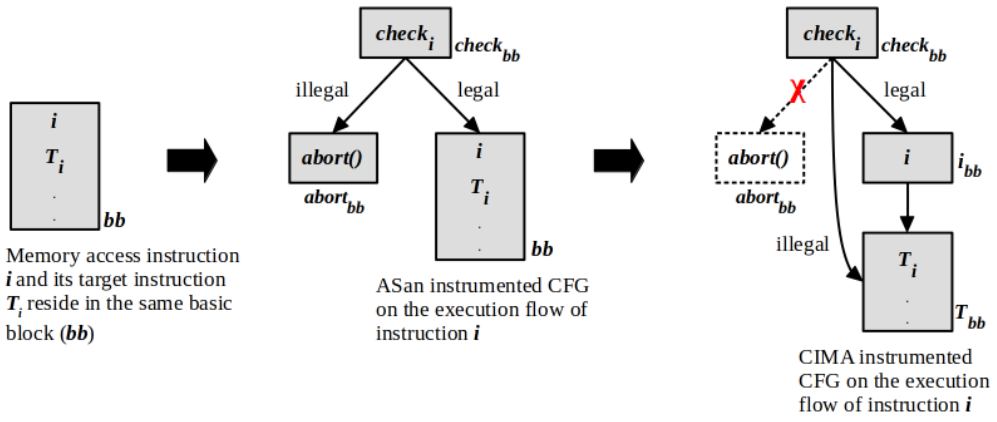

i
T_i
bb
Memory access instruction i and its target instruction T_i reside in the same basic block (bb)
check_i
check_bb
illegal
legal
abort()
abort_bb
i
T_i
bb
ASan instrumented CFG on the execution flow of instruction i
check_i
check_bb
legal
abort()
abort_bb
i
i_bb
illegal
T_i
T_bb
CIMA instrumented CFG on the execution flow of instruction i

Fig. 6: Construction of the CFG when i and T_i reside in the same basic-block.

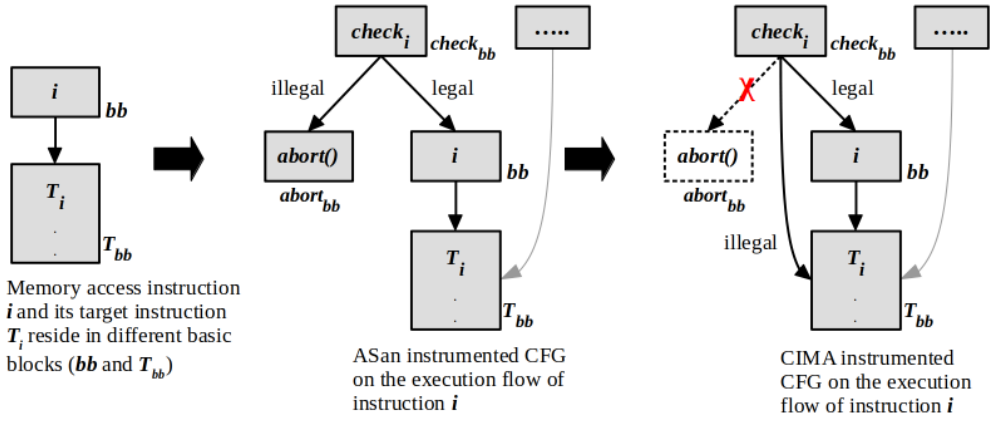

i
bb
T_i
T_bb
Memory access instruction i and its target instruction T_i reside in different basic blocks (bb and T_bb)
check_i
check_bb
.....
illegal
legal
abort()
abort_bb
i
bb
T_i
T_bb
ASan instrumented CFG on the execution flow of instruction i
check_i
check_bb
.....
legal
abort()
abort_bb
i
bb
illegal
T_i
T_bb
CIMA instrumented CFG on the execution flow of instruction i

Fig. 7: Construction of the CFG when i and T_i reside in different basic-blocks.

instruction i is detected as illegal. The newly constructed CFG in this scenario is illustrated in Figure 6.

Scenario 2: When i and T_i are resided in different basic-blocks. In this case, we do not need to split the basic-block since i and T_i are resided in their respective basic-blocks in the original CFG. So, control can simply jump to the target basic-block (i.e. T_{bb}) when instruction i is detected as illegal. The CFG construction of this scenario is depicted in Figure 7.

A more discussion of the two scenarios as well as a detailed account of CIMA can be found in [9, 10].

Validation Although CIMA effectively mitigates memory-safety attacks, there exists semantics-preserving issues as skipping instructions affects the original program semantics. A careful discussion of such issue is provided in the original paper [9]. From the discussion, there are only a few corner cases (which might not even happen in a real-world) where CIMA might not be effective (see in[9]). Apart from that, CIMA does not affect the execution of the program and this is validated experimentally. A detailed discussion of CIMA's validation can be found in [9].

3.3 KASan: kernel-space detection

Overview Nowadays, most PLCs are user-mode applications running on POSIX-like OSs, such as Linux OS [8]. For instance, Allen Bradley PLC5 uses *Microware OS-9* [46]; Siemens SIMATIC[49] uses *Microsoft Windows*; Schneider Quantum uses *VxWorks* [46]; Emerson DeltaV uses *VxWorks* [46]; and OpenPLC uses *Linux OS* [40].

Therefore, in addition to the exploitation threats at user-space, cyberattacks may also exploit memory-safety vulnerabilities that could be found in the underlying operating systems hosting the PLCs. In particular, attacks may exceptionally target vulnerabilities in the Linux kernel (as recent trends also show in CVE [38]). Therefore, the underlying OS kernel of the PLC is another attack surface for memory-safety attacks targeting CPS.

To address the kernel-space security concern, we ported KASan in CPS by fixing its various architectural incompatibility issues. KASan [30] is a fast and dynamic memory error detector tool designed for the Linux kernel.

KASan is also a code-instrumentation based tool and it follows a similar approach with ASan to detect memory-safety violations. However, with the assumption of not to heavily affect performance of the Linux kernel, KASan is made to cover only a limited (but critical) set of memory-safety vulnerabilities, such as buffer overflows, double-free errors and use-after-free bugs. Consequently, its runtime overhead is considerably lower when compared with that of ASan.

KASan also follows a similar mitigation strategy with that of ASan. It automatically aborts the victim program when a memory violation is detected, which is an ineffective and non-resilient mitigation strategy as discussed earlier.

Porting KASan in CPS Porting KASan to our CPS setup was not a straightforward task because of architectural incompatibility and other technical issues. The current version of KASan is designed only for the x86_64 and ARM64 target architectures [30, 8]. Unfortunately, Raspbian (i.e. the underlying operating system of Raspberry PI) is a 32-bit OS, and no support for the 64-bit architecture so far. Hence, the Raspbian kernel (at the time of writing this paper) is based on a 32-bit (i.e. ARM32) target architecture. For this reason, it is not possible to directly enforce KASan to the Raspbian kernel. Besides, we also encountered several technical difficulties when the kernel-level building tool (i.e. GCC) is different from the user-level GCC (which incorporates ASan and CIMA).

To overcome these technical problems, it was essential to build a custom Linux kernel with ARM64 architecture. To accomplish this, first we built a cross-compiler toolchain (using ASan and CIMA enabled GCC) on a 64-bit Linux OS. Then, we cross-compiled a custom Raspberry PI Linux kernel (with 64-bit architecture) using our cross-compiler toolchain. Finally, we managed to compile the custom Linux kernel with KASan (by enabling the `KASAN=y` compiler switch and other configuration flags). This hardened Linux kernel can then detect kernel-level memory-safety attacks or violations, such as buffer overflows, use-after-free bugs and double-free memory errors.

Similar to the cases in ASan and CIMA, our experimental setup has also allowed us to empirically measure the performance overhead of KASan. Then, we quantify its performance impact and hence acceptability in the CPS context.

4 Experimental design

The effectiveness of our proposed security measures against memory-safety attacks is experimented on realistic CPS testbeds. This section presents a brief discussion of the two CPS testbeds used to conduct our experiments.

4.1 SWaT

SWaT [33] 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 briefly discuss the purification process in SWaT and how we reproduce SWaT to conduct our experiments.

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 The overall purification process of SWaT is shown in Figure 8. A detailed account of SWaT can be found in [9, 33].

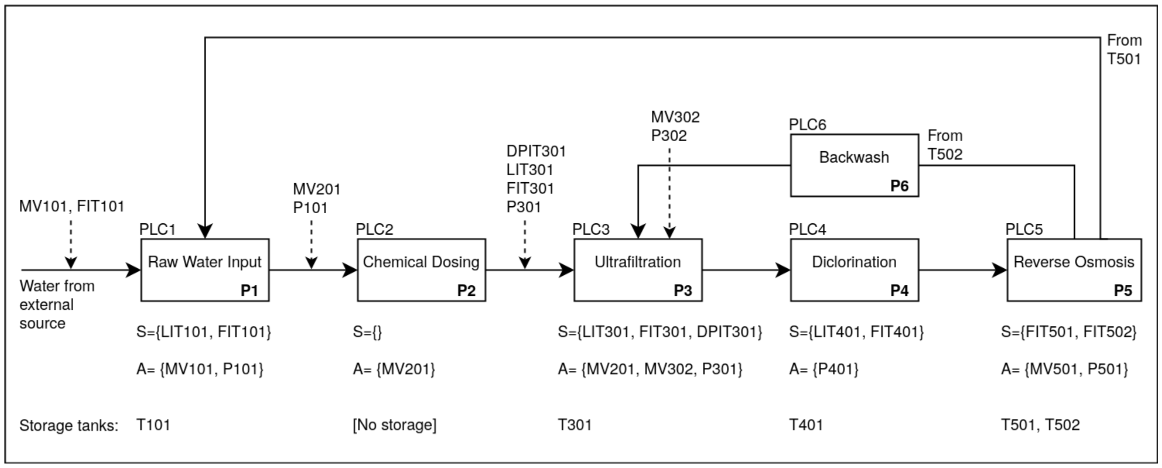


Fig. 8: Overview of the water purification process in SWaT.
[**Definition of the acronyms:** S = Sensor, A = Actuator, T = Tank, P = Process, MV = Motorized Valve, LIT = Level Indicator Transmitter, FIT = Flow Indicator Transmitter, DPIT = Differential Pressure Indicator Transmitter]

Open-SWaT SWaT is designed using proprietary PLCs. Hence, it is not possible to directly enforce our memory-safety since we cannot modify the firmware of these PLCs. To address this problem, we designed an open testbed, named Open-SWaT.

Open-SWaT [9, 8] is a mini CPS we designed using open-source PLCs [40] by mimicking the features and operational details of SWaT. For example, it mimics the hardware specifications of the SWaT

PLCs (e.g. a CPU speed of 200MHz and a user memory of 2MB), a Remote Input/Output (RIO) terminal (containing 32 digital inputs (DI), 16 digital outputs (DO)), 13 analog inputs (AI), the real-time constraints, the PLC program (containing 129 instructions), the communication frequencies and the full SCADA system. A high-level architecture of Open-SWaT is illustrated in Figure 9. A detailed account of Open-SWaT can also be found in [9].

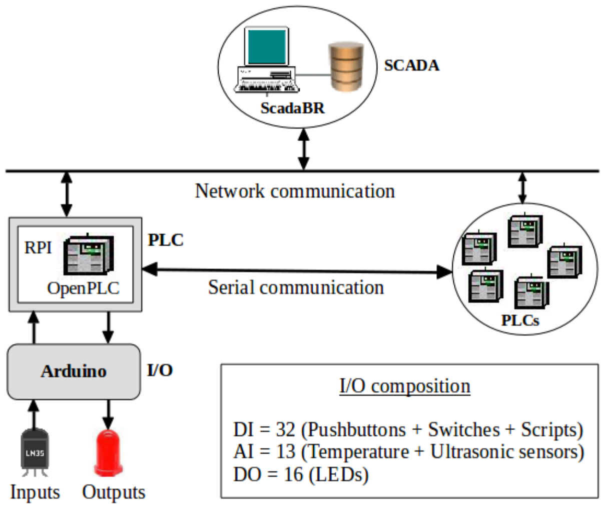


Fig. 9: Architecture of Open-SWaT

4.2 SecUTS

SecUTS [55] is a CPS testbed designed to secure a Metro SCADA system. A detailed account of SecUTS testbed can be found in [55, 9].

Unfortunately, this testbed is also based on proprietary PLCs, hence we cannot directly enforce our full-stack memory-safety to these PLCs. Consequently, we also prototyped Open-SecUTS (by mimicking the SecUTS testbed) using the OpenPLC controller. It comprises 6 DI (emergency and control buttons) and 9 DO (tunnel and station lightings, ventilation and alarms). Subsequently, we enforced our full-stack memory-safety to Open-SecUTS and evaluated its practical applicability in a Metro SCADA system.

5 Evaluation and discussion

In this section, we evaluate and discuss the experimental results of our full-stack memory-safety enforcement in CPS. In particular, we evaluated the security guarantee (i.e. the detection and mitigation accuracy), the efficiency (i.e. tolerability of the overall overhead in CPS), the mitigation resilience and the memory usage overheads of the full-stack enforcement.

5.1 Security guarantees

In this section, as a sanity check on our full-stack memory-safety enforcement, we evaluate the detection and mitigation accuracy of ASan, CIMA and KASan over various memory-safety vulnerabilities. As discussed in previous sections, ASan and KASan may rarely miss some memory violations, such as global buffer overflows and use-after-free bugs, if an attacker manages to corrupt regions outside the redzones. Due to this reason, ASan and KASan could give rare false positives on these violations. Apart from this, ASan and KASan effectively detects memory-safety violations. CIMA, on the other hand, accurately mitigates all the memory-safety violations detected by ASan, but only in user-space. Table 1 summarizes the detection and mitigation coverage of the three tools over various memory-safety vulnerabilities.

Table 1: Detection and mitigation accuracy of the full-stack memory-safety.

Vulnerabilities	ASan (Detection)		CIMA (Mitigation)		KASan (Detection)	
	False positive	False negative	False positive	False negative	False positive	False negative
Stack buffer overflow	No	No	No	No	No	No
Heap buffer overflow	No	No	No	No	No	No
Global buffer overflow	No	Rarely[*]	No	No	No	No
Use-after-free bugs	No	Rarely[*]	No	No	No	No
Use-after-return bugs	No	No	No	No	Uncovered	Uncovered
Initialization order bugs	No	No	No	No	Uncovered	Uncovered
Memory leaks	No	No	No	No	Partially	Partially
Double-free errors	No	No	No	No	No	No
Uninitialized memory reads	Uncovered	Uncovered	Uncovered	Uncovered	Uncovered	Uncovered

[*] The reasons are discussed in Section 5.1 and further details can also be found in the original paper [47].

5.2 Performance

In this section, we discuss the practical tolerability of the overall full-stack memory-safety overhead, i.e., ASan + CIMA + KASan ($\hat{T_s''}$), in CPS. A detailed performance report of our full-stack memory-safety enforcement, including the overhead contributed by each tool, is depicted in Table 2 (for Open-SWaT) and Table 3 (for Open-SecUTS). According to the results, $\hat{T_s''}$ is 91.02% (for Open-SWaT) and 85.49% (for Open-SecUTS).

Subsequently, we evaluate tolerability of the full-stack memory-safety overhead both in the average-case and worst-case scenarios. Essentially, we have checked if the overall overhead satisfies the conditions defined on Eq. (2) (for average-case scenario) and Eq. (3) (for worst-case scenario).

First, we evaluated tolerability of the overhead in the average-case scenario. For Open-SWaT, $mean(\hat{T_s''}) = 522.39\mu s$ (cf. Table 2), and $T_c = 10000\mu s$; and for Open-SecUTS, $mean(\hat{T_s''}) = 470.91\mu s$ (cf. Table 3), and $T_c = 30000\mu s$. Therefore, according to Eq. (2), the overhead is tolerable with a large magnitude for both SWaT and SecUTS in the average-case scenario (see the tolerability bar in Figure 10a and 10c, respectively).

Similarly, we evaluated tolerability of the full-stack overhead in the worst-case scenario. For Open-SWaT, $max(\hat{T_s''}) = 4342.27\mu s$ (cf. Table 2), and $T_c = 10000\mu s$; and for Open-SecUTS, $max(\hat{T_s''}) = 4124.57\mu s$ (cf. Table 3), and $T_c = 30000\mu s$. Both overheads satisfy Eq. (3), hence the overall overhead is tolerable for both SWaT and SecUTS even in the worst-case scenario (see the tolerability bar in Figure 10b and 10d, respectively). That means, the overhead of our full-stack memory-safety largely meets the real-time constraints of SWaT and SecUTS both in the average-case and worst-case scenarios, while significantly increasing its security.

Table 2: MSO of the full-stack memory-safety for the Open-SWaT Testbed.

Operations	Number of cycles	Nodes	CPU MHz	Original (T_s)		ASan ($\hat{T_s}$)				ASan + CIMA ($\hat{T_s'}$)				ASan + CIMA + KASan ($\hat{T_s''}$)			
				Mean (in µs)	Max (in µs)	Mean (in µs)	Max (in µs)	MSO (in µs)	MSO (in %)	Mean (in µs)	Max (in µs)	MSO (in µs)	MSO (in %)	Mean (in µs)	Max (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	135.35	1648.48	75.97	127.94
Execution	50000	6	200	69.09	611.82	115.88	720.36	46.79	67.72	118.97	802.18	49.88	72.2	120.39	912.81	53.3	74.25
Output	50000	6	200	145.01	981.09	185.37	1125.45	40.36	27.83	199.89	1213.62	54.88	37.85	266.65	1780.98	121.64	83.88
Total	50000	6	200	273.48	2381.03	419.69	2978.13	146.21	53.46	441.72	3167.15	168.24	61.52	522.39	4342.27	248.91	91.02

Table 3: MSO of the full-stack memory-safety for the Open-SecUTS Testbed.

Operations	Number of cycles	Nodes	CPU MHz	Original (T_s)		ASan ($\hat{T_s}$)				ASan + CIMA ($\hat{T_s'}$)				ASan + CIMA + KASan ($\hat{T_s''}$)			
				Mean (in µs)	Max (in µs)	Mean (in µs)	Max (in µs)	MSO (in µs)	MSO (in %)	Mean (in µs)	Max (in µs)	MSO (in µs)	MSO (in %)	Mean (in µs)	Max (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	120.29	1624.47	60.45	101.02
Execution	50000	1	200	48.56	488.38	91.36	443.61	42.8	88.14	104.41	676.19	55.85	115.01	106.89	827.75	58.33	120.12
Output	50000	1	200	145.47	850.62	175.59	1045.34	30.12	20.71	178.91	924.11	33.44	22.99	243.73	1672.35	98.26	67.55
Total	50000	1	200	253.87	2078.94	381.83	2390.96	127.96	50.4	398.39	2506.39	144.52	56.93	470.91	4124.57	217.04	85.49

5.3 Resilience

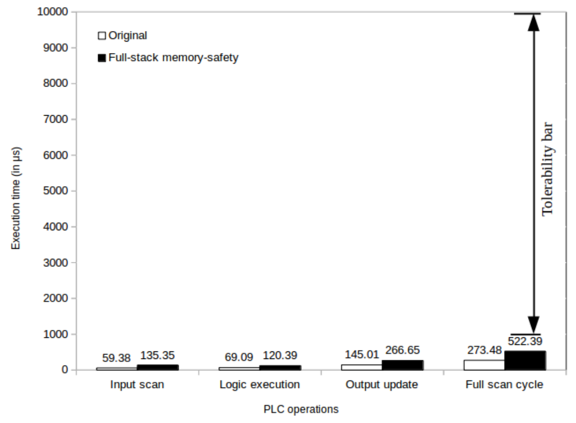
As discussed in Section 3.1, ASan simply aborts the victim program when a memory-safety violation or an attack is detected. Hence, it does not satisfy the physical-state resiliency requirement since the control delay τ is indefinite. However, the introduction of CIMA overcomes this mitigation limitation of ASan. To assess the physical-state resiliency of CIMA, we need to check if the control delay τ, caused by the MSO or the mitigation strategy, satisfies Eq. (8). In the former case, we already showed in the preceding section that the overall MSO induced by our full-stack memory-safety is tolerable, i.e., $\hat{T_s} \leq T_c$. Hence, the induced MSO does not affect the physical-state resiliency. In the later case, since CIMA does not abort or restart the PLC when mitigating memory-safety violations or attacks, it does not render system unavailability. That means, the control delay τ caused by our mitigation strategy is zero, hence Eq. (8) is satisfied. Therefore, the physical-state resiliency constraint is satisfied for our user-space memory-safety enforcement.

However, as discussed in Section 3.3, there is no a resilient mitigation strategy for the kernel-space since KASan simply aborts the victim program upon detection of memory-safety attacks or violations. As discussed, such mitigation strategy leads to system unavailability and hence not acceptable in CPS. Therefore, addressing this problem is left as a future work.

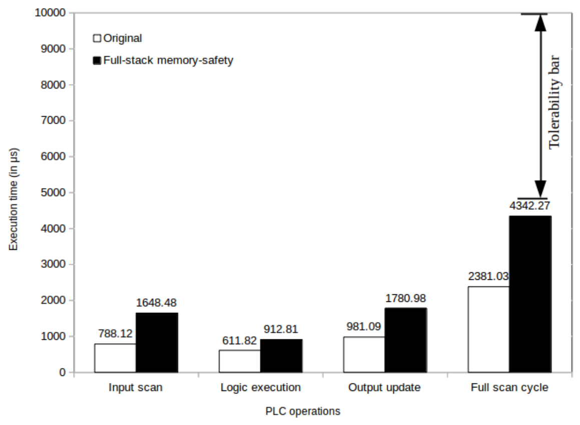
5.4 Memory usage overheads

The memory usage overhead of our full-stack memory-safety is also measured and evaluated. Table 4 and 5 summarize the virtual memory, real memory, binary size and shared library usages for the Open-SWaT and Open-SecUTS testbeds, respectively.

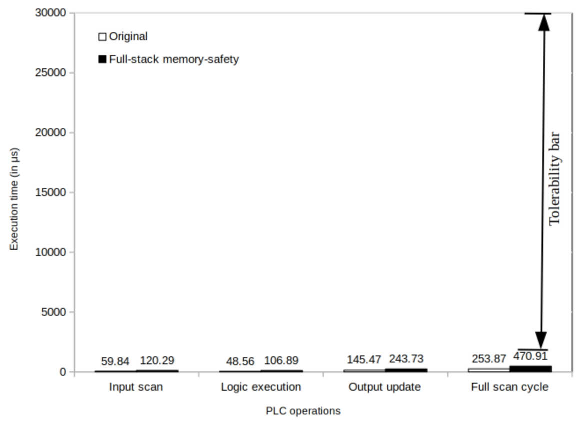
We notice a significant increase in virtual memory usages (11.11× for Open-SWaT and 10.92× for Open-SecUTS) in our full-stack enforcement. This is mainly due to the large redzones created with *malloc* as part of the ASan and KASan approaches. However, the real memory overhead is only 1.4× (for Open-SWaT) and 1.21× (for Open-SecUTS). We believe these overheads are still acceptable considering that most PLCs nowadays come with a minimum of 1GB memory size. Moreover, this memory overhead is an acceptable tradeoff in the light of strong countermeasures provided by our full-stack memory-safety solution. Finally, we observe that majority of the memory-usage overhead is incurred by ASan while KASan and CIMA only introduce a minimal and negligible memory usage overheads, respectively.



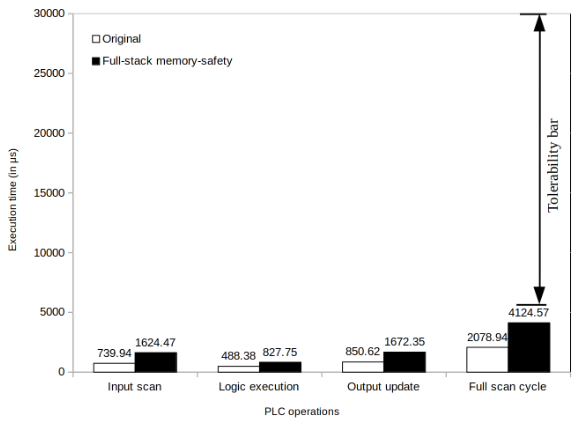
(a) The average-case MSO for Open-SWaT



(b) The worst-case MSO for Open-SWaT



(c) The average-case MSO for Open-SecUTS



(d) The worst-case MSO for Open-SecUTS

Fig. 10: Tolerability of the full-stack MSO for the Open-SWaT and Open-SecUTS testbeds.

6 Related work

Cyber-Physical Systems and Memory Safety Previously, we proposed to study the tolerability of a secure compilation in the context of CPS [7]. In that work, we consider only memory error detection at user-space, does not consider mitigation resiliency against availability attacks and has a preliminary evaluation on a simulated testbed. In our other work [8], we consider both kernel and user space detection, but without considering any mitigation strategy and also disregarding availability attacks. In our recent work [9], we study dynamic instrumentation for achieving resilience against availability attacks based on memory-safety violations, but without detection of kernel-space attacks. In this work, we consider a full-stack memory safe compilation that is also resilient to memory-safety attacks in user-space.

Generic memory safety countermeasures Softbound [34] and its extension CETS [35] offer a high-level memory-safety. However, these tools induce a very high runtime overhead (116%), which might not be tolerable in CPS. Moreover, no mitigation is implemented in Softbound and CETS, hence no protection

Table 4: Memory usage overheads of the full-stack memory-safety for Open-SWaT.

Category	Original	ASan		ASan+CIMA		ASan+CIMA+KASan	
		Instrumented	Overhead	Instrumented	Overhead	Instrumented	Overhead
Virtual memory usage	62.97MB	549.38MB	8.72×	557.5MB	8.85×	699.91MB	11.11×
Real memory usage	8.17MB	10.31MB	1.26×	11.2MB	1.37×	13.26MB	1.40×
Binary size	144KB	316KB	2.19×	324KB	2.25×	324KB	2.25×
Shared library size	3196KB	4288KB	1.34×	4288KB	1.34×	4288KB	1.34×

Table 5: Memory usage overheads of the full-stack memory-safety for Open-SecUTS.

Category	Original	ASan		ASan+CIMA		ASan+CIMA+KASan	
		Instrumented	Overhead	Instrumented	Overhead	Instrumented	Overhead
Virtual memory usage	56.37MB	489.29MB	8.68×	490.6MB	8.70×	615.33MB	10.92×
Real memory usage	8.76MB	9.81MB	1.12×	10.21MB	1.17×	10.62MB	1.21×
Binary size	136KB	288KB	2.12×	296KB	2.18×	296	2.18×
Shared library size	3196KB	4288KB	1.34×	4288KB	1.34×	4288	1.34×

against availability attacks. SafeCode[21] is also a compile-time based memory-safety tool that operates at the source code level. It instruments *load* and *store* instructions to prevent illegal memory accesses. However, SafeCode failed to prevent direct stack overflows toward function pointers and static arrays defined in many library functions such as fscanf(), sscanf(), sprintf() and snprintf().

Countermeasures based on control-flow integrity (CFI) A number of CFI-based solutions (e.g. [1, 54, 53, 26]) have been developed to prevent execution flow redirection attacks. However, these solutions have the following limitations in general: *(i)* determining the required CFG (often via a static analysis) is very difficult and requires a significant amount of memory; *(ii)* data-oriented attacks [28], which do not divert the execution flow, cannot be detected; *(iii)* finally, these solutions do not implement mitigation strategies against the attacks. Consequently, the applicability of CFI-based solutions is limited in a CPS environment.

Memory safety and availability Rinard et al. [43] implemented "failure-oblivious computing" that allows a vulnerable program to continue its execution even in the presence of memory errors. This is accomplished via the following techniques.Systematically fabricated values are returned for invalid memory reads, and all invalid memory writes are simply discarded. But, this approach has several limitations. Firstly, providing fabricated values to the invalid memory reads might not be always acceptable since it could result in an undesirable outcomes to the system. Secondly, the "failure-oblivious computing" approach is designed only against buffer-overflow vulnerabilities, hence other critical vulnerabilities, such as dangling pointers and memory leaks, are not covered. Finally, this approach was designed only for Servers and Desktop computers and its applicability in the context of CPS is not validated.

In summary, to the best of our knowledge, there is no any prior works that develops and evaluates full-stack memory-safety in the light of hard real-time constraints and physical-state resiliency imposed in CPS.

7 Conclusion

In this research, we explored the applicability of strong countermeasures against memory-safety attacks in CPS, covering both the user-space and kernel-space attack surfaces. Moreover, we enforced a resilient mitigation strategy with a focus on availability. In particular, to evaluate efficiency of our proposed full-stack countermeasure, the induced performance overhead (both the average-case and worst-case overhead) is evaluated against the real-time constraints of the two CPS under test.

As our experimental results revealed, the proposed full-stack countermeasure is efficient and effective enough in detecting and mitigating memory-safety attacks in a CPS environment. As a compile-time tool, our full-stack memory-safety enforcement is dependent on the availability of the source code. Therefore, binary instrumentation with such solutions can be considered as a future work. A resilient mitigation strategy for the kernel-space is also left as a future work.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the ACM Conference on Computer and Communications Security. CCS'05 (2005)
2. Armando, A., Carbone, R., Chekole, E.G., Petrazzuolo, C., Ranalli, A., Ranise, S.: Selective release of smart metering data in multi-domain smart grids. In: Cuellar, J. (ed.) Smart Grid Security. pp. 48–62. Springer International Publishing (2014)
3. Armando, A., Carbone, R., Chekole, E.G., Ranise, S.: Attribute based access control for apis in spring security. In: Proceedings of the 19th ACM Symposium on Access Control Models and Technologies. pp. 85–88. SACMAT'14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2613087.2613109, http://doi.acm.org/10.1145/2613087.2613109
4. Berger, E.D., Zorn, B.G.: Diehard: Probabilistic memory safety for unsafe languages. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'06 (2006)
5. Bittau, A., Belay, A., Mashtizadeh, A., Maziéres, D., Boneh, D.: Hacking blind. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy. SP'14 (2014)
6. Burow, N., McKee, D., Carr, S.A., Payer, M.: Cup: Comprehensive user-space protection for c/c++. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. ASIACCS'18 (2018)
7. Chekole, E.G., Castellanos, J.H., Ochoa, M., Yau, D.K.Y.: Enforcing memory safety in cyber-physical systems. In: Katsikas S. et al. (eds) Computer Security. SECPRE 2017, CyberICPS 2017 (2017), https://doi.org/10.1007/978-3-319-72817-9_9
8. Chekole, E.G., Chattopadhyay, S., Ochoa, M., Huaqun, G.: Enforcing full-stack memory safety in cyber-physical systems. In: Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS'18) (2018), https://doi.org/10.1007/978-3-319-94496-8_2
9. Chekole, E.G., Chattopadhyay, S., Ochoa, M., Guo, H., Cheramangalath, U.: Cima: Compiler-enforced resilience against memory safety attacks in cyber-physical systems. Computers & Security p. 101832 (2020). https://doi.org/10.1016/j.cose.2020.101832
10. Chekole, E.G., Cheramangalath, U., Chattopadhyay, S., Ochoa, M., Guo, H.: Taming the war in memory: A resilient mitigation strategy against memory safety attacks in cps. CoRR **abs/1809.07477** (2018), https://arxiv.org/abs/1809.07477
11. Chekole, E.G., Huaqun, G.: Ics-sea: Formally modeling the conflicting design constraints in ics. In: Proceedings of the Fifth Annual Industrial Control System Security (ICSS) Workshop. p. 60–69. ICSS (2019). https://doi.org/10.1145/3372318.3372325
12. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems. pp. 205–218. SenSys'07 (2007)
13. CVE-2011-5007: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5007 (2011)
14. CVE-2012-0929: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0929 (2012)
15. CVE-2012-6436: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6436 (2012)
16. CVE-2012-6438: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6438 (2012)
17. CVE-2013-0674: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0674 (2013)
18. CVE-2015-1449: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1449 (2015)
19. CVE-2015-7937: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7937 (2015)
20. CVE-2016-5814: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5814 (2016)

21. Dhurjati, D., Kowshik, S., Adve, V.: Safecode: enforcing alias analysis for weakly typed languages. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. PLDI'06 (2006)
22. F. Ch. Eigler: Mudflap: pointer use checking for C/C++. In: GCC Developer's Summit. Red Hat Inc. (2003)
23. Follner, A., Bodden, E.: Ropocop — dynamic mitigation of code-reuse attacks. Journal of Information Security and Applications (2016)
24. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08). CCS'08 (2008)
25. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI'03 (2003)
26. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: 2016 IEEE European Symposium on Security and Privacy (2016)
27. Giraldo, J., Urbina, D., Cardenas, A., Valente, J., Faisal, M., Ruths, J., Tippenhauer, N.O., Sandberg, H., Candell, R.: A survey of physics-based attack detection in cyber-physical systems. ACM Comput. Surv. (2018)
28. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. IEEE Symposium on Security and Privacy (2016)
29. Jang, D., Tatlock, Z., Lerner, S.: Safedispatch: Securing c virtual calls from memory corruption attacks. In: the Network and Distributed System Security Symposium. NDSS'14 (2014)
30. KASAN: The kernel address sanitizer. https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html (2018)
31. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems - A Cyber-Physical Systems Approach. LeeSeshia.org, second edition, version 2.0 edn. (2015)
32. Lee, E.A.: Cyber physical systems: Design challenges. In: International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) (2008)
33. Mathur, A.P., Tippenhauer, N.O.: Swat: a water treatment testbed for research and training on ics security. In: 2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater). pp. 31–36 (2016). https://doi.org/10.1109/CySWater.2016.7469060
34. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for C. In: Proceedings of the SIGPLAN conference on Programming language design and implementation. PLDI'09 (2009)
35. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Cets: Compiler enforced temporal safety for C. In: the International Symposium on Memory Management (2010)
36. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. (2005)
37. Novark, G., Berger, E.D.: Dieharder: Securing the heap. In: In Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 573–584. CCS'10 (2010)
38. (NVD), N.V.D.: Nvd statistics on the linux kernel vulnerabilities. https://nvd.nist.gov/vuln/search/results?adv_search=false&form_type=basic&results_type=overview&search_type=all&query=linux+kernel (2018)
39. Olowononi, F.O., Rawat, D.B., Liu, C.: Resilient machine learning for networked cyber physical systems: A survey for machine learning security to securing machine learning for cps. IEEE Communications Surveys & Tutorials pp. 1–1 (2020). https://doi.org/10.1109/COMST.2020.3036778
40. OpenPLC: Openplc. http://www.openplcproject.com/ (2018)
41. project, T.D.: http://deputy.cs.berkeley.edu (2007)
42. github repository, A.: Comparison of addresssanitizer with other memory safety tools. https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools (2015)
43. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebee, Jr., W.S.: 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)
44. Saito, T., Watanabe, R., Kondo, S., Sugawara, S., Yokoyama, M.: A survey of prevention/mitigation against memory corruption attacks. In: International Conference on Network-Based Information Systems (NBiS) (2016)

45. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c applications. In: S&P'15 (2015)
46. SECURITY, T.: https://www.tofinosecurity.com/blog/plc-security-risk-controller-operating-systems/
47. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12) (2012)
48. Sha, L., Gopalakrishnan, S., Liu, X., Wang, Q.: Cyber-physical systems: A new frontier. In: IEEE international conference on sensor networks, ubiquitous, and trustworthy computing (SUTC'08) (2008)
49. Siemens-AG: https://new.siemens.com/global/en/products/automation/systems/industrial/plc.html (2018)
50. Simpson, M.S., Barua, R.K.: Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. Software: Practice and Experience **43**(1) (2013)
51. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: 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 (2013)
52. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. IEEE S&P (2013)
53. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in gcc & llvm. In: Proceedings of the 23rd USENIX Security Symposium. USENIX'14 (2014)
54. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Proceedings of the USENIX Security Symposium. USENIX'13 (2013)
55. Zhou, L., Guo, H., Li, D., Wong, J.W., Zhou, J.: Mind the gap: Security analysis of metro platform screen door system. In: Proceedings of the Singapore Cyber-Security RandD Conference (SG-CRC'17) (2017)