Enforcing Memory Safety in Cyber-Physical Systems

Eyasu Gethun Chekole[1,2], John Henry Castellanos[1], Martín Ochoa[1], and David K. Y. Yau[1,2]

[1] Singapore University of Technology and Design
[2] Advanced Digital Sciences Center, Singapore

Abstract. Cyber-Physical Systems (CPS) integrate computations and communications with physical processes and are being widely adopted in various application areas. However, the increasing prevalence of cyber attacks targeting them poses a growing security concern. In particular, attacks exploiting memory-safety vulnerabilities constitute a major attack vector against CPS, because embedded systems often rely on unsafe but fast programming languages to meet their hard time constraints. A wide range of countermeasures has been developed to provide protection against these attacks. However, the most reliable countermeasures incur in high runtime overheads. In this work, we explore the applicability of strong countermeasures against memory-safety attacks in the context of realistic Industrial Control Systems (ICS). To this end, we design an experimental setup, based on a secure water treatment plant (SWaT) to empirically measure the memory safety overhead (MSO) caused by memory-safe compilation of the Programmable Logic Controller (PLC). We then quantify the tolerability of this overhead in terms of the expected real-time constraints of SWaT. Our results show high effectiveness of the security measure in detecting memory-safety violations and a MSO (197.86µs per scan-cycle) that is also tolerable for the SWaT simulation. We also discuss how different parameters impact the execution time of PLCs and the resulting absolute MSO.

1 Introduction

Cyber-physical systems [1–3], which integrate computations and communications with physical processes, are gaining attention and being widely adopted in various application areas including power grid, water systems, transportation, manufacturing, healthcare services and robotics, among others. Despite their importance, the increasing prevalence of cyber attacks targeting them poses a serious security risk. On the other hand, real-time requirements and legacy hardware/software limit the practicality of certain security solutions available. Thus, the trade-off between security, performance and cost remains one of the main design challenges for CPS.

Cyber attacks in CPS can target either the communication network or the individual network nodes. Network attacks might sniff, drop, or compromise data

packets as they traverse the communication channels, e.g., man-in-the-middle (MITM) and replay attacks. They exploit vulnerabilities in the communication paths or protocols. Attacks against computing nodes, e.g., PLCs, might exploit vulnerabilities in the firmware or control software. For example, malware can corrupt the memory of the PLC to hijack or otherwise subvert its operations. In this paper, we focus on the class of software/firmware attacks [4, 5] that compromises the integrity of the PLC's memory system to inject or trigger malicious code or data.

Memory safety vulnerabilities arise due to the use of programming languages where memory is handled manually such as C/C++. Those languages are popular and they are particularly relevant in systems with stringent real-time constraints since they allow skilled programmers to produce very efficient compiled code. However, programmers are also responsible to avoid potential security flaws in their code. Since firmwares of PLCs are commonly implemented in these languages, the memory unsafety represents a significant security concern. Real-world examples, such as buffer overflows and dereferences of dangling pointers, are regularly discovered and reported in modern PLCs.

For instance, Common Vulnerabilities and Exposures (CVE) [6] have been reported for a wide range of memory safety vulnerabilities on PLCs in the last couple of decades. For example, a buffer overflow vulnerability reported by CVE concerns Allen-Bradley's RSLogix Micro Starter Lite (CVE-2016-5814) [7], which allows remote attackers to execute arbitrary codes via a crafted rich site on summary (RSS) project file. Yet other buffer overflow vulnerabilities are reported on [8] and [9] on this PLC. Similarly, a CVE has also recently reported memory safety vulnerabilities discovered on Siemens PLC ([10], [11]), Schneider Electric Modicon PLC ([12], [13]), ABB PLC automation ([14]), and so on.

Since such attacks regularly impact general IT systems as well, a wide range of countermeasures have been developed against memory-safety attacks. They differ by various characteristics including architecture, runtime overhead, type of memory errors covered, accuracy of detecting errors, platforms supported, etc. One can generally classify them into four categories: *probabilistic* (e.g., stack canaries, address space layout randomization (ASLR), position independent execution (PIE), [15, 16]), *paging-based* (e.g., non-executable memory page (NX)), *control-flow integrity (CFI) based* (e.g., [17–19]), and *code-instrumentation based* (e.g., ASan [20], SoftBoundCETS[21, 22], [23–26]).

Most of the countermeasures mentioned above are designed for desktop computers and servers with x86-based target architectures. When applied in a CPS context, they have the following limitations. First, almost all of them have architectural compatibility problems in working with RISC-based ARM or AVR CPU architectures. More fundamentally, many countermeasures have non-negligible runtime overheads, which might be prohibitive in the context of CPS. Hence, vis-a-vis the exploitation concerns, performance and availability are equally critical in a CPS environment.

Thus, to cover a wide range of memory safety violations, the code instrumentation based countermeasures, which we refer to as memory safety tools, offer

stronger guarantees. These tools do not directly counter memory-safety attacks. Rather, they detect and mitigate memory safety violations before the attackers get a chance to exploit them. Numerous memory safety tools are available that differ by error coverage, accuracy, and performance. Although there are published benchmarks for the overheads caused by such tools, which give an intuition of average penalties to be paid when using them, it is still unclear how they perform in a CPS context.

In this research, after reviewing several available tools, we port a popular memory-safety compilation tool (ASan [20]) to work on a system architecture mimicking a realistic CPS. We adopt an empirical approach to measure its MSO, so that we can quantify its performance impact and hence acceptability for different applications. Our experiments are inspired by SWaT, a realistic CPS water system testbed that contains a set of real-world vendor-supplied PLCs. However, the vendor's PLC firmware is closed-source; it does not allow us to incorporate additional memory safety solutions. Hence, we prototyped an experimental setup, which we call open-SWaT, based on open-source PLCs to mimic the behavior of the SWaT according to its detailed operational profile. We report experiments conducted on open-SWaT, which indicate that the MSO would not impact the normal operation of SWaT.

In summary, this work tackles the problem of *quantifying the practical tolerability of strong memory-safety enforcement on realistic Cyber-Physical Systems with hard real-time constraints and limited computational power.*

We make the following contributions: **a)** We enforce a memory safety countermeasure based on secure compiling for a realistic CPS environment. **b)** We empirically measure the tolerability of the induced overhead of the countermeasure based on the constraints of a real industrial control system. **c)** We discuss parameters that affect the absolute overheads (in terms of time) in order to generalize our observations on tolerability beyond our case study.

2 Background

In this section, we provide background information on cyber-physical systems, the CPS testbed we use for experimenting (SWaT) and the memory safety tool we enforce to our CPS design (ASan).

2.1 Overview of CPS

Unlike traditional IT systems, CPS involve complex interactions among various entities while integrating physical plants and control devices (PLCs) via communication networks. These interactions are also constrained by hard deadlines. Missing deadlines could result in disruption of control loop stability or a complete system failure, in the worst case. This makes CPS highly real-time constrained systems. CPS are also highly resource-constrained systems. Edge devices, e.g., PLCs and input/output devices, have limited memory size and CPU speed. In general, as shown on Figure 1, CPS consists of the following entities:

-*Plants*: entities where physical processes take place.
-*Sensors*: devices that observe or measure state information of plants and physical processes, which will be used as inputs for PLCs.
-*PLCs*: entities that make decisions and issue control commands (based on inputs obtained from sensors) to control plants.
-*Actuators*: entities that implement the control commands issued by PLCs.
-*Communication networks*: communication medias where packets (containing sensor measurements, control commands, alarms, diagnostic information, etc) transmit over from one entity to another.
-*SCADA*: an entity designed for process controlling and monitoring. It consists of human-machine interface (HMI) – for displaying state information of plants and historian server – for storing all operating data, alarm history and events.

2.2 Overview of SWaT

SWaT is a fully operational water purification plant designed for research in the design of secure cyber-physical systems. It produces 5 gallons/minute of doubly filtered water.

Purification process. The whole water purification process is carried out by six distinct, but cooperative, sub-processes. Each process is controlled by an independent PLC (details can be found on [27]).

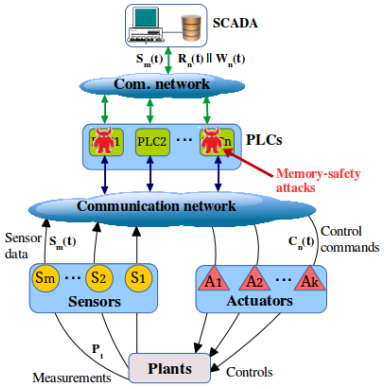

Fig. 1. The CPS architecture and memory-safety attacks

Components and specifications. The design of SWaT consists of the following components and system specifications:
-*PLCs*: six redundancy closed-source Allen Bradley ControlLogix L5571 (1756-L71) PLCs are deployed to control each sub-process. They communicate one another via EtherNet/IP-CIP (Common Industrial Protocol).
-*Real-time constraint*: the real-time constraint of SWaT is 10ms. The notion of real-time constraint (in the context of CPS) is discussed in detail on Section 3.5.
-*Remote input/output (RIO)*: remote terminals consisting of digital inputs (DI), analog inputs (AI) and digital outputs (DO). RIO of SWaT contains 32 DI (water level and pressure switches), 13 AI (water pressure, flow rate and water level sensors), and 16 DO (actuators such as pumps and motorized valves).
-*Communication frequency*: the six PLCs exchange packets among each other and with the SCADA system depending on operational conditions. If we take the busiest PLC (PLC2), it transfers as high rate as 382 packets per second with its most active peer (PLC3) or as low as three packets per second to another. Regarding connections with all devices in SWaT, we estimate that the busiest PLC (PLC2) sends and receives requests to a ratio of 1000 packets per second.
-*PLC program*: a complex control logic written in ladder language generally. It consists of various instructions (see the list on Table 1). PLC2 (the busiest PLC) has a complex PLC program consisting of 127 instructions.

-*SCADA system*: a touch panel, an HMI, is mounted to SWaT to provide users a local system supervisory monitoring and controls.

2.3 ASan

As discussed in the introduction, despite several memory safety tools being available, there applicability in the CPS environment is limited due to compatibility and performance reasons. After researching and experimenting on various memory safety tools, we chose ASan [20] as a basis for our empirical study because of its error coverage, high detection accuracy and relatively low runtime overhead.

ASan is a compile-time code instrumentation memory safety tool. It inserts memory safety checks into the program code at compile-time, and it detects and mitigates memory safety violations at runtime. ASan covers several memory safety vulnerabilities such as buffer overflows, dangling pointers (use after free), use after return, memory leaks and initialization order bugs. Although there are also some memory errors, e.g., uninitialized memory reads, that are not covered by ASan, such errors are less critical and rarely exploited in practice.

ASan has relatively low overhead when compared to other code instrumentation tools. Different level of performance optimization options are also available for the tool. A detailed account on error coverage and runtime overhead of ASan (in comparison with other tools) is provided on [20].

Similar to other memory safety tools, the off-the-shelf ASan has compatibility issues with RISC-based ARM or AVR based architectures. ASan has also a problem of dynamically linking shared libraries, e.g., `glibc`, for our experimental setup. Therefore, as explained on Section 4.2, our initial task was fixing those problems to fit our experimental design. For this task it was crucial that ASan is an open-source project, which allowed for several customizations.

3 System and attacker model

We have seen high-level overview of CPS and its architecture on Section 2.1. In this section, we model the interactions of various entities in the CPS, attack scenarios targeting them, and MSO and its tolerability for a given CPS.

3.1 Modeling interactions

Entities of CPS (as shown on Figure 1) interact one another in a continuous and loop-back manner. Suppose the state vector of plant P at time t is $P_t \in \mathbb{R}^k$, where k is dimensions of state variables of the plant such as water level, water flow, water pressure, etc. Sensor S_m measures the state information $S_m(t) \in \mathbb{R}$ at time t and sends to PLCs. When PLC_n receives the state information $S_m(t)$ from sensor S_m, it makes decision and issues control command $C_n(t) \in \mathbb{R}^q$, where q is dimensions of control variables for different actuators such as pumps, valves, etc, in accordance with its control logic. Thus, $C_n(t) = f(S_m(t))$, where f is the control logic of PLC_n. Then, it sends this control command to actuators.

Actuators will take actions to implement the command. The actions taken by the actuators will change the state information of the plant. The loop-back routine will continue in a similar way.

The SCADA system directly communicates with PLCs via a specific communication protocol, e.g., ENIP, Modbus, and CIP. It can send read command ($R_n(t) \in \mathbb{R}$) or write command ($W_n(t) \in \mathbb{R}$) to PLC_n at time t. With $R_n(t)$, it requests state information of the plant. PLCs will then feed it with the information received from sensors. The SCADA system will then display the state information to users via its integrated HMI. With $W_n(t)$, (users via) the SCADA system can also issue control commands to change state information of the plant. Upon receiving, PLCs will request actuators to implement it. The whole CPS process is a continuous process triggered and synchronized by system time.

3.2 Attacker model

As discussed in the introduction, memory-safety attacks such as code injection and code reuse mainly exploit memory safety vulnerabilities on firmwares of the PLCs such as buffer overflows, dangling pointers, and so on. Their main goal is to take control of PLC's programs, and eventually affect the CPS. Bittau et al. [28] provided detail information on how to exploit memory safety vulnerabilities, develop an attack technique and then take control of the vulnerable system. In general, there are five main steps involved in the memory-safety attacks scenario.

1. Interacting with the PLC, e.g., via network connection (for remote attacks).
2. Finding a memory safety vulnerability in the firmware/control software so that will be exploited accordingly, e.g., buffer overflow vulnerability.
3. Triggering a memory safety violation to happen on the PLC at runtime, e.g., overflow the buffer.
4. Overwriting important addresses of the vulnerable program, e.g., overwrite return address of the PLC program.
5. Using the new return address, diverting control flow of the program to an injected (malicious) code (in case of code injection attacks) or to existing modules of the vulnerable program (in case of code reuse attacks). In the former case, the attacker can get control of the PLC with its injected code. In the later case, the attacker still needs to collect important gadgets from the program (basically by scanning the program's text segment), then he will synthesis a shellcode that will allow him to get control of the PLC.

After getting control of the PLC, the attacker would able to make changes on normal operations of the PLC. For example, by changing the control logic he can force the PLC to issue tainted control command $C'_n(t)$.

$$C'_n(t) = f'(S_m(t)), f \neq f' \tag{1}$$

where f' is the tainted control logic by the attacker. Actuators will be then forced to implement the control command issued by the attacker. This will change the operational behavior of the PLC, and so does the CPS, in general. Figure 1 shows an architectural point of view of memory-safety attacks in CPS.

3.3 Modeling system performance

Overall performance of CPS can be affected by communication latencies such as latencies of packets carrying sensor measurements ($L_{S_m(t)} \in \mathbb{R}$) and latencies of packets carrying control commands ($L_{C_n(t)} \in \mathbb{R}$). In addition, it can be also affected by the execution time that will be taken by each entity to complete its respective tasks. Sensors take some execution time ($T_{sensors} \in \mathbb{R}$) to measure or observe state information of plants. PLCs also take substantial time ($T_{PLCs} \in \mathbb{R}$) to make decisions and issue control commands that involves complex cyclic operations (details on Section 3.4). Similarly, actuators take some execution time ($T_{actuators} \in \mathbb{R}$) to implement control commands. Therefore, the overall CPS time ($T_{CPS} \in \mathbb{R}$) can be intuitively represented with the following formula:

$$T_{CPS} = L_{S_m(t)} + L_{C_n(t)} + T_{sensors} + T_{PLCs} + T_{actuators} \tag{2}$$

However, accurately modeling the overall performance of the dynamics of CPS is not that much intuitive since the interactions among the entities are very complex and some of them are non-deterministic. As described on Section 5.4, it can be also affected by various internal and external factors. Zhang et al. [29] modeled an approximation of the dynamics of CPS as a general difference equations with state variables, control variables, and the noises. However, we do not need to go in detail on this since the main focus of this paper is modeling memory safety overheads rather than overall performance of CPS. Therefore, we will mainly focus on T_{PLCs} on the following section since it is directly involved in the MSO computation.

3.4 Modeling MSO

To ensure memory safety, firmwares of PLCs should be compiled with a memory safety tool. Hence the memory safety overhead will be added to the execution time of PLCs, i.e., T_{PLCs}. To model T_{PLCs}, we need to clearly understand how PLCs operate in CPS. PLCs handle two main processes – a communication process and a scan cycle process (the terms are proposed by the authors in the context of the conducted research). These processes are handled by two separate and parallel threads – a communication thread and a scan cycle thread, respectively. The communication thread handles any network communication related tasks, e.g., creating connections with communicating entities, receiving and sending network requests, etc. It is a continuous process. On the other hand, the scan cycle thread handles the main PLC process that involves three operations: scan inputs, execute control program and update outputs. The PLC scan cycle starts by reading state of all inputs from sensors and storing them to the PLC input buffer. Then, it will execute the PLC's control logic and issue control commands according to the state of sensor inputs. The scan cycle will be then concluded by updating output values to the output buffer and sending control commands to the actuators. Unlike the communication process, the PLC scan process is a cyclic process.

Before modeling MSO, it is important to describe the following notions that involve in the overall control process:

- *Cycle time ($T_c \in \mathbb{R}$)*: an upper bound time set for a scan cycle, i.e., a scan cycle must be completed within this time period. When the cycle time is over, the next scan cycle will start immediately.
- *Scan time ($T_s \in \mathbb{R}$)*: the measurement of the actual time elapsed by the PLC scan cycle process.

$$T_s = T_{si} + T_{ep} + T_{uo}, \tag{3}$$

  where $T_{si} \in \mathbb{R}$ is time elapsed to scan inputs, $T_{ep} \in \mathbb{R}$ is time elapsed to execute PLC program and $T_{uo} \in \mathbb{R}$ is time elapsed to update outputs.
- *Buffer waiting time ($T_{bw} \in \mathbb{R}$)*: the communication and scan cycle threads can run in parallel, but they access shared resources, i.e., input and output buffers. When the two threads try to access those shared buffers, race condition or deadlock could happen. To avoid that happening, a mutual exclusion buffer locking function is introduced to PLCs that prevents any attempt of simultaneous access to the shared buffers. This will introduce some waiting time. Suppose (T_{bw}) is the waiting time of the scan cycle thread. Computing T_{bw} is non-deterministic; because the locking process is not deterministic since it involves a stochastic communication process. This buffer waiting time will directly affect the PLC scan time. Therefore, the overall PLC scan time (T_s) can be rewritten as follows,

$$T_s = T_{si} + T_{ep} + T_{uo} + T_{bw} \tag{4}$$

- *Thread sleeping time ($T_{ts} \in \mathbb{R}$)*: if the scan cycle is completed before the allocated cycle time, the thread will sleep for the remaining cycle time.

By design, $T_s + T_{ts} \leq T_c$ unless otherwise execution time of the PLC is affected by external factors, e.g., MSO. The memory safety overhead ($MSO \in \mathbb{R}$) is the average difference in scan time for a PLC firmware that is compiled with and without a memory safety tool. As we will see in the following, using average time is justified by the small standard deviation in multiple measurements.

$$MSO = \hat{T_s} - T_s, \tag{5}$$

where $\hat{T_s}$ is the scan time when the firmware is compiled with memory-safety.

Computing MSO is also non-deterministic since the scan time (T_s or $\hat{T_s}$) computation is non-deterministic because of T_{bw}. Obviously, non-deterministic computations are less precise as compare to actual measurements. Thus, given the fact that the real-time constraints of CPS are in the order of milliseconds (often $3 - 10ms$), there would no better approach than the empirical one to precisely measure MSO and quantify its tolerability. That is the main reason why we proposed the empirical approach for this research.

3.5 Quantifying tolerability

We define MSO tolerability with respect to real-time constraints of CPS. Real-time constraints of CPS often defined based on the real-time constraints of its respective PLCs. Real-time constraint of PLCs is defined with its respective cycle time. As discussed on Section 3.4, an upper bound time, i.e., T_c, will be set for each scan cycle. Meaning, each scan cycle of the PLC must be completed within the cycle time duration specified for that PLC, i.e., $T_s \leq T_c$. This constraint is called *real-time constraint* of the PLC.

But, as we discussed above, the memory safety overhead increases the PLC scan time. If the scan time with memory safety enabled, i.e., $\hat{T_s}$, still respects real-time constraint of the PLC, then the memory safety overhead can be considered as *tolerable*. Therefore, we define tolerability of MSO in average-case scenario ($T(MSO)$), i.e., when tolerability is quantified based on an average $\hat{T_s}$ of n scan cycle measurements, as follows,

$$T(MSO) = \begin{cases} \text{Tolerable}, & \text{if } mean(\hat{T_s}) \leq T_c \\ \text{Not tolerable}, & \text{otherwise} \end{cases} \tag{6}$$

where $mean(\hat{T_s}) = \frac{\sum_{i=1}^{n} \hat{T_s}(i)}{n}$, where $\hat{T_s}(i)$ denotes measurement of the i-th scan cycle performed with a memory-safe compilation and n is the total number of scan cycles performed.

We often use the average-case scenario to quantify tolerability. However, a single scan time could potentially violate timing constraint of the PLC. Therefore, it is also important to consider the worst-case scenario, i.e., with the maximum $\hat{T_s}$ of n measurements, to validate if the MSO is fully tolerable. Thus, tolerability of the MSO in the worst-case scenario ($T(WMSO)$) can be defined as,

$$T(WMSO) = \begin{cases} \text{Tolerable}, & \text{if } max(\hat{T_s}) \leq T_c \\ \text{Not tolerable}, & \text{otherwise} \end{cases} \tag{7}$$

where $max(\hat{T_s}) = max(\{\hat{T_s}(1), ..., \hat{T_s}(n)\})$.

The choice of scenarios may depend on delay-sensitivity of the CPS or users' preference. For SWaT, we use the average-case as a basis to validate tolerability. But, we also use the worst-case to validate if it is fully tolerable. If it is not, then we do further investigations to figure out if the root cause is actually the MSO or other exceptions, e.g., sudden service interruptions due to unforeseen reasons.

4 Experimental design

As discussed in the introduction, SWaT is based on closed-source PLCs. Thus, we designed open-SWaT – a CPS based on open source PLCs that mimics features and behaviors of SWaT. We discuss design details of open-SWaT and our experimental results in the following.

4.1 open-SWaT

open-SWaT is designed using OpenPLC [30] – an open source PLC for industrial control systems. With open-SWaT, we reproduce operational details of SWaT; in particular we reproduce the main characteristics (mentioned on Section 5.4) that have a significant impact on the scan time and MSO:

PLCs: we designed the PLCs using an OpenPLC controller that runs on top of Linux on Raspberry PI. To reproduce hardware specifications of SWaT PLCs, we specified 200MHz fixed CPU speed and 2Mb user memory for our PLCs.

RIO: we use Arduino Mega as RIO terminal. It has AVR based processor with 16MHz clock speed. To reproduce the number of I/O devices of SWaT, we used 32 DI (push-buttons, switches and scripts), 13 AI (temperature and ultrasonic sensors) and 16 DO (light emitter diodes (LEDs)).

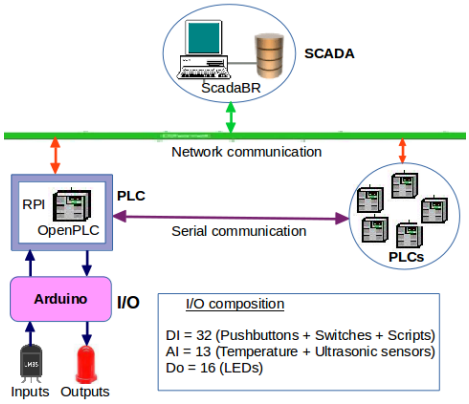

Fig. 2. Architecture of open-SWaT

PLC program: we designed a control logic in ladder diagram that has similar complexity to the one in SWaT (a sample diagram is shown on Figure 3). It consists of various types of instructions such as logical (AND, OR, NOT, SR (set-reset latch)), arithmetic (addition (ADD), multiplication (MUL)), comparisons (equal (EQ), greater than (GT), less than (LT), less than or equal (LE)), counters (up-counter (CTU)), timers (turn on timer (TON), turn off timer (TOF)), contacts and coils (normally-open (NO), normally-closed (NC)). The overall PLC program consists of 129 instructions (see details on Table 1).

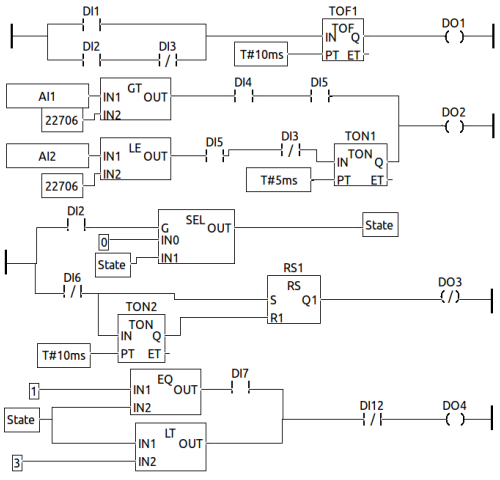

Fig. 3. Sample PLC program in ladder diagram

Table 1. Instruction count

Instructions	Count
Logical	
AND	17
OR	14
NOT	5
SR	1
Arithmetic	
ADD	1
MUL	2
Comparisons	
EQ	3
GT	3
LT	2
LE	2
Timers	
TON	3
TOF	9
Counters	
CTU	1
Selections	
SEL	1
MAX	1
Contacts	
NO	38
NC	3
Coils	
NO	21
NC	2
Total	129

Communication frequency: A high-level architecture of open-SWaT is shown on Figure 2. open-SWaT uses both type of modbus communication protocols – modbus TCP (for Ethernet or wireless communication) and modbus RTU (for serial communication). Frequency of communication among PLCs and the SCADA system is similar to that in SWaT. The communication between PLCs and Arduino is via USB serial communication. The frequency of receiving inputs and sending outputs with Arduino is 100Hz.

Real-time constraints: based on the real-time constraint of SWaT, we set a 10ms cycle time (real-time constraint) to each PLC in open-SWaT.

SCADA system: we use ScadaBR [31], a full SCADA system consisting of a web-based HMI.

In summary, the design of open-SWaT is, by design, very close to SWaT. In particular, the inputs PLCs receive from sensors, the logic they execute, the number of nodes they are communicating with, the frequency of communications, etc, are designed to be similar. Thus, we can expect that the MSO in open-SWaT would also remain close to that in SWaT. Therefore, if the MSO is tolerable in open-SWaT, it would be the same for SWaT. In the future, as a further validation step, we may replace some PLCs at SWaT with the open-source and memory-safety enabled PLCs of open-SWaT to evaluate its functional equivalence.

4.2 Memory safety compilation

As stated on the introduction, our approach to counter memory-safety attacks is by secure compiling of the PLCs' firmware with a compile-time code-instrumentation tool. We ported ASan for that, but porting ASan to our CPS design was not a straightforward task because of its compatibility and dynamic library linking problems. Thus, we fixed those problems by modifying and rebuilding its source code and by enabling dynamic library linking runtime options.

To do secure compilation, we also need to integrate ASan with a native C/C++ compiler. Fortunately, ASan can work with GCC or CLANG with a `-fsanitize=address` switch – a compiler flag that enables ASan at compile time. Various types of compile-time and runtime flags (e.g., performance optimization, stack trace, diagnostics, specific error checking and blacklist file) are also available. Therefore, we compiled our PLC using GCC with ASan enabled.

4.3 Detection and mitigation

As discussed on 2.3, ASan instruments the protected program to ensure that memory access instructions never read or write the so called "poisoned" redzones [20]. Redzones are small regions of memory inserted in between any two stack, heap or global objects. Since the program should never address them, access to them indicates an illegal behavior and it will be considered as memory-safety violation. This policy detects sequential buffer over/underflows, and some of the more sophisticated pointer corruption bugs such as dangling pointers (use after

free) and use after return bugs (see the full list on Table 3). With this enforcement, we detected two global buffer overflow vulnerabilities on the OpenPLC Modbus implementation.

ASan's mitigation approach is based on the principle of "automatically aborting" the vulnerable program whenever a memory-safety violation is detected. It is effective in restricting memory-safety attacks not to able to exploit the vulnerabilities. However, this might not be acceptable in a CPS environment since the abortion affects system availability, and potentially controllability of a process. Thus, we are currently working on advanced mitigation strategies to address these limitations.

4.4 Experimental results

We have implemented a function using POSIX clocks (in nanosecond resolution) that measures execution time of each operation in the PLC scan cycle. Table 2 summarizes overall performance of the PLC including execution time of each PLC operation and its respective memory safety overheads. For statistical analysis, we have also included minimum (min), maximum (max), mean, variance and standard deviation (sd) of each operation.

Table 2. Memory safety overhead (MSO)

Operations	Number of scan cycles	Network devices	T_s (in µs)	$\hat{T}_s$ (in µs)					Average MSO	
			mean	min	max	mean	var	sd	in µs	in %
Input scan	10x1000	6	50.657	69.740	1997.348	112.380	8587.425	92.668	61.723	121.85
Program execution	10x1000	6	79.537	104.323	1325.732	176.723	7799.361	88.314	91.19	122.19
Output update	10x1000	6	111.635	107.552	1518.909	150.583	5503.488	74.185	38.95	34.89
Full scan time	10x1000	6	241.829	281.615	4841.989	439.686	7296.758	85.056	197.86	81.82

5 Evaluation and discussion

After conducting experiments, we performed a detailed evaluation to figure out whether the memory safety tool is accurate enough to detect memory safety violations and efficient enough to work in a CPS environment. In brief, our evaluation has three parts: *security (accuracy)* – detection accuracy of ASan, *performance (efficiency)* – tolerability of its runtime overhead in CPS, and *memory usage* overheads.

5.1 Security

As a sanity check on our configuration, we have evaluated our setup against the memory safety vulnerabilities listed on Table 3, to explore detection accuracy of ASan in the CPS environment. The results show, as in the original paper [20], the tool detects

Table 3. Detection accuracy

Vulnerabilities	False positive	False negative
Stack buffer overflow	No	No
Heap buffer overflow	No	No
Global buffer overflow	No	Rare
Dangling pointers	No	Rare
Use after return	No	No
Initialization order bugs	No	No
Memory leaks	No	No

memory safety violations with high accuracy – with no false positives for all vulnerabilities, and very few false negatives for global buffer overflows and use after return vulnerabilities. The paper [20] briefly discussed the conditions how and where the false negatives could happen. According to [20], ASan performs better than the other memory safety tools in terms of error coverage and accuracy of detecting them. ASan also effectively mitigates the detected violations regardless of the limitations discussed on Section 4.3.

5.2 Performance

According to published benchmarks [20], the average runtime overhead of ASan is about 73%. However, all measurements were taken on a non-CPS environment. In our setting, the average overhead is 197.86µs (81.82%) as shown on Table 2. To validate its tolerability for our system, i.e., SWaT, we have checked if it satisfies the tolerability conditions defined on Equation (6) (for average-case) and Equation 7 (for worst-case). As demonstrated on Table 2, $mean(\hat{T}_s) = 439.686$µs, and $T_c = 10000$µs. Therefore, according to Equation (6), the MSO is tolerable for SWaT with the average-case scenario.

To validate the tolerability in the worst-case scenario, we check if it satisfies Equation 7. As shown on Table 2, $max(\hat{T}_s) = 4841.989$µs, and $T_c = 10000$µs. It is still tolerable, thus ASan satisfies the real-time constraint of SWaT both in the average-case and worst-case scenarios. Therefore, we conclude that SWaT would fully tolerate the overhead caused by memory-safe compilation, while significantly increasing its security.

5.3 Memory usage

Table 4. Memory overheads (in MB)

Category	Original	Instrumented	Increase
Memory usage	22.516	852.480	37.86×
Stack size	0.136	0.142	1.04×
Binary size	0.140	0.328	2.34×
Shared library usage	2.796	3.372	1.21×

We also evaluated memory usage overheads of our security measure. Table 4 summarizes the increase in memory usage, stack size, binary size and shared library usage collected by reading VmPeak, VmStk, VmExe and VmLib fields, respectively, from `/proc/self/status`. There is significant increase in memory usage due to the allocation of large redzones with `malloc`. This overhead is still acceptable since most PLCs come with at least 1GB memory size.

5.4 Sensitivity analysis

There are many factors, that can affect the scan time or MSO of a PLC. But, as we verified it experimentally, the main variables that significantly affect the scan time and MSO are (i) the number of input devices (sensors) connected to the PLC ($N_S \in \mathbb{N}$) (ii) the number of output devices (actuators) connected to the PLC ($N_A \in \mathbb{N}$) (iii) complexity of the PLC program ($C_P \in \mathbb{R}^z$, where z = {number of instructions, type of instructions}) (iv) frequency of communications (frequency of packets) between the PLC and other entities ($F_P \in \mathbb{R}$)

and (v) CPU speed of the PLC ($S_{CPU} \in \mathbb{R}$). F_P is dependent on the number of connections the PLC is communicating with. We have done a preliminary sensitivity analysis on these variables regarding its effect on the PLC scan time and MSO. The variables affect execution time of the PLC operations discussed on Section 3.4. N_S affects input scan time, i.e., T_{si}, of the PLC. N_A affects output update time, i.e., T_{uo}, of the PLC. As discussed on Section 4.1, the PLC program consists of various type of instructions. Each instruction has different execution time. Therefore, C_P can be determined by the number and the type of instructions it consists of; and it affects the program execution time, i.e., T_{ep}. F_P affects the buffer waiting time, i.e., T_{bw}, hence it affects T_{si}, T_{ep} and T_{uo} (or T_s in general). S_{CPU} affects all operations of the PLC.

Table 5. Sensitivity analysis

Logic complexity	Num. of scan cycles	CPU speed (in MHz)	Num. of sensors	Number of actuators	Num. of connections	Average T_s (in µs)	Average $\hat{T}_s$ (in µs)	Average MSO	
								in µs	in %
Simple program (containing 4 instructions)	10x1000	200	2	2	0	190.74	292.26	101.52	53.22
					2	208.67	318.33	109.66	52.55
					4	203.72	330.41	126.69	62.19
					6	213.65	351.47	137.82	64.51
Complex program (containing 129 instructions)	10x1000	200	45	16	0	222.83	339.55	116.72	52.38
					2	232.15	376.12	143.97	62.02
					4	236.84	420.47	183.63	77.53
					6	241.83	439.69	197.86	81.82

Because of space limitation, we present here only the preliminary experiment we have done by involving two kinds of PLC logic: a simple PLC program containing 4 instructions (interacting with 2 sensors and 2 actuators) and a more complex program consisting of 129 instructions (interacting with 45 digital and analog sensors and 16 actuators), for different networking configurations (0 to 6 connections) to explore the impact of each variable in the scan time and the MSO computation. Our experimental result is shown on Table 5.

6 Related work

In this section, we explore related works done in providing memory safety solutions against memory-safety attacks and measuring and analyzing memory safety over heads in the CPS environment.

As discussed in the introduction, many memory safety tools are available designed for general IT systems. However, they have limitations to work in the CPS environment. SoftBoundCETS is a compile-time code-instrumentation tool that detects wide range of spatial memory safety (SoftBound [21]) and temporal memory safety (CETS [22]) violations in C. However, its runtime overhead is very high (116%) as compare to ASan (73%). In addition, it is incompatible for the CPS environment; because it is implemented only for the x86-64 target architecture and it is also dependent on the LLVM infrastructure.

Cooprider et al. [32] enforced efficient memory safety solution for TinyOS applications by integrating Deputy [33], an annotation based type and memory safety compiler, with nesC [34], a C compiler. Thus, they managed to detect

memory safety violations with high accuracy. To make this memory safety solution practical in terms of CPU and memory usage, they did aggressive optimization by implementing a static analyzer and optimizer tool, called cXprop. With cXprop, they managed to reduce memory safety overhead of Deputy from 24% to 5.2%, and they also improved memory usage through dead code elimination. However, their solution has limitations to apply it in a CPS environment, because it is dependent on runtime libraries of TinyOS.

Zhang et al. [29] modeled the trade-off between privacy and performance in CPS. While they leveraged the differential privacy approach to preserve privacy of CPS, they also analyzed and modeled its performance overhead. They proposed an approach that optimizes the system performance while preserving privacy of CPS. This work is interesting from point of view of analyzing performance overheads in CPS, but not from the memory safety perspective.

Stefanov et al. [35] proposed a new model and platform for the SCADA system of an integrated CPS. With the proposed platform, they modeled real-time supervision of CPS, performance of CPS based on communication latencies, and also he assessed communication and cyber security of the SCADA system. He followed a generic approach to assess and control various aspects of the CPS. However, he did not specifically work on memory-safety attacks or MSO.

Several CFI based solutions (e.g., [18], [19]) have been also developed against memory safety attacks. However, CFI based solutions have some limitations in general (i) determining the required control flow graph (often using static analysis) is hard and requires a significant amount of memory; (ii) attacks that do not divert control flow of the program cannot be detected (for instance using Data Oriented attacks [36]). These and other reasons can limit the applicability of CFI solutions in the CPS environment.

In summary, to the best of our knowledge, there is no prior research work done that enforced memory safety solutions specifically to the CPS environment, and that measured and analyzed tolerability of memory safety overheads in accordance to real-time constraints of cyber-physical systems.

7 Conclusion

In this work we presented the results of implementing a strong memory safety enforcement in a simulated albeit realistic industrial control system using ASan. Our setup allowed us to benchmark and empirically measure the runtime overhead of the enforcement and, based on the real-time constraints of an ICS, to judge the applicability in a realistic scenario. Our experiments show that the real-time constraint of SWaT can be largely met even when implementing strong memory safety countermeasures in realistic hardware. We also preliminary discuss what factors impact the performance of such a system, in a first attempt to generalize our results.

In the future, we intend to study other CPS with different constraints, e.g., in power grid systems, water distribution or smart home devices. Such studies will

allow us to extrapolate formulas predicting the tolerability of systems to MSO and thus aiding in the design of resilient CPS before such systems are deployed.

References

1. Sha, L., Gopalakrishnan, S., Liu, X., Wang, Q.: Cyber-Physical Systems: A New Frontier. In: SUTC'08. (2008)
2. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems - A Cyber-Physical Systems Approach. Second edition, version 2.0 edn. (2015)
3. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: ISORC'08
4. Basnight, Z., Butts, J., Lopez, J., Dube, T.: Firmware modification attacks on programmable logic controllers. IJCIP **6**(2) (2013) 76 – 88
5. Cui, A., Costello, M., Stolfo, S.J.: When firmware modifications attack: A case study of embedded exploitation. In: NDSS'13. (2013)
6. MITRE: Common Vulnerabilities and Exposures. https://cve.mitre.org/
7. CVE-5814. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5814
8. CVE-6438. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6438
9. CVE-6436. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6436
10. CVE-0674. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0674
11. CVE-1449. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1449
12. CVE-0929. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0929
13. CVE-7937. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7937
14. CVE-5007. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5007
15. Berger, E.D., Zorn, B.G.: Diehard: Probabilistic memory safety for unsafe languages. In: PLDI'06. (2006)
16. Novark, G., Berger, E.D.: Dieharder: Securing the heap. In: CCS'10. (2010)
17. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: CCS'05. (2005) 340–353
18. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: USENIX'13
19. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in gcc & llvm. In: USENIX'14. (2014) 941–955
20. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: USENIX ATC'12. (2012)
21. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: Highly compatible and complete spatial memory safety for C. In: PLDI'09
22. Nagarakate, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: Compiler enforced temporal safety for C. In: ISMM'10. (2010)
23. 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) 93–128
24. Bruening, D., Zhao, Q.: Practical Memory Checking with Dr. Memory. In: CGO'11
25. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe Retrofitting of Legacy Software. ACM Trans. Program. Lang. Syst. **27**(3) (2005)
26. Frank Ch. Eigler: Mudflap: pointer use checking for C/C++. Red Hat Inc.
27. Ahmed, C.M., Adepu, S., Mathur, A.: Limitations of state estimation based cyber attack detection schemes in industrial control systems. In: SCSP-W 2016. (2016)
28. Bittau, A., Belay, A., Mashtizadeh, A., Maziéres, D., Boneh, D.: Hacking blind. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy. (2014)

29. Zhang, H., Shu, Y., Cheng, P., Chen, J.: Privacy and performance trade-off in cyber-physical systems. IEEE Network **30**(2) (2016) 62–66
30. OpenPLC. http://www.openplcproject.com/
31. ScadaBR. http://www.scadabr.com.br/
32. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: SenSys'07. (2007) 205–218
33. The Deputy project: . http://deputy.cs.berkeley.edu (2007)
34. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: PLDI'03. (2003)
35. Stefanov, A., Liu, C.C., Govindarasu, M., Wu, S.S.: Scada modeling for performance and vulnerability assessment of integrated cyber-physical systems. International Transactions on Electrical Energy Systems **25**(3) (2015) 498–519
36. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. SP'16 (2016)