Enforcing Full-Stack Memory-Safety in Cyber-Physical Systems

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

[1] Singapore University of Technology and Design, Singapore, Singapore
[2] Institute for Infocomm Research (I^{2}R), Singapore, Singapore
[3] Department of Applied Mathematics and Computer Science, Universidad del Rosario, Bogotá, Colombia

Abstract. Memory-safety attacks are one of the most critical threats against Cyber-Physical Systems (CPS). As opposed to mainstream systems, CPS often impose stringent timing constraints. Given such timing constraints, how can we protect CPS from memory-safety attacks? In this paper, we propose a full-stack memory-safety attack detection method to address this challenge. We also quantify the notion of tolerability of memory-safety overheads (MSO) in terms of the expected real-time constraints of a typical CPS. We implemented and evaluated our proposed solution on a real-world Secure Water Treatment (SWaT) testbed. Concretely, we show that our proposed solution incurs a memory-safety overhead of 419.91µs, which is tolerable for the real-time constraints imposed by the SWaT system. Additionally, We also discuss how different parameters of a typical CPS will impact the execution time of the CPS computational logic and memory safety overhead.

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, two major issues have raised concerns about the safety of CPS in general. On the one hand, the increasing prevalence of cyber attacks 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.

In this paper, we focus on memory-safety attacks against computing nodes of a CPS. These attacks typically are launched on programmable logic controllers (PLCs) and exploit memory-safety vulnerabilities. Most PLCs nowadays are user-mode applications running on top of a POSIX-like OS, often Linux OS. Therefore, memory-safety vulnerabilities may be discovered on the PLC firmware and control software (*user-space*) or the Linux kernel (*kernel-space*).

For example, a malware can corrupt the memory of the PLC or the kernel to hijack or otherwise subvert its operations.

Memory-safety vulnerabilities arise due to the use of programming languages where memory management is handled manually, such as C/C++. Those languages are particularly relevant in systems with stringent real-time constraints since they allow skilled programmers to produce efficient compiled code. However, since firmwares of PLCs and operating systems are commonly implemented in memory-unsafe languages (for the sake of efficiency), the memory unsafety remains a significant security concern. For instance, buffer overflows and dangling pointers, are regularly discovered and reported in modern PLCs.

Common Vulnerabilities and Exposures (CVE) [4] have been reported for a wide-range of memory-safety vulnerabilities only on PLCs for the last couple of decades. For example, a buffer overflow vulnerability concerns Allen-Bradley's RSLogix Micro Starter Lite (CVE-2016-5814) [5]. This allows remote attackers to execute arbitrary code via a crafted rich site on summary (RSS) project file. Yet other buffer overflow vulnerabilities are reported on this PLC [6, 7]. Similarly, CVEs are also recently reported for memory-safety vulnerabilities discovered on Siemens PLC ([8], [9]), Schneider Electric Modicon PLC ([10], [11]), ABB PLC automation ([12]), and so on. Recent CVE reports also show a high volume of interest in exploiting the Linux kernel [13].

Existing countermeasures against memory-safety attacks [14–26] face several challenges to be employed in the context of CPS. First, almost all of them have architectural compatibility problems in working with PLCs, because the PLCs are often based on RISC-based ARM or AVR CPU architectures. More fundamentally, the countermeasures have non-negligible runtime overheads, which may unacceptably compromise the performance of a CPS. Violation of timing constraints in a CPS may lead to serious consequences, including complete system damage or disruption and incorrect control by the use of stale information. Hence, vis-a-vis the exploitation concerns, performance and availability are equally critical in a CPS environment.

To cover a wide range of memory-safety errors, the code-instrumentation based countermeasures, which we refer to as memory-safety tools, offer stronger guarantees. These tools detect memory-safety violations before the attackers get a chance to exploit them. 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 paper, we leverage memory-safety compilation tools ASan [20] (for the user-space) and KASan [27] (for the kernel-space) to enforce full-stack memory safety in CPS. We quantify the performance impact of our solution via an empirical approach that measures the memory-safety overhead. We evaluated our approach on SWaT [28], a realistic CPS water treatment testbed that contains a set of real-world vendor-supplied PLCs. However, the PLC firmware for the SWaT is closed-source and hence, it does not allow us to incorporate additional memory-safety solutions. To circumvent this challenge, we prototyped an experimental setup, which we call open-SWaT, based on open-source PLCs and mimic

the behavior of the SWaT according to its detailed operational profile. Our experiments on open-SWaT reveal that the introduced memory-safety overhead would not impact the normal operation of SWaT.

In summary, this work tackles the problem of *quantifying the practical tolerability of a strong full-stack 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 full-stack memory-safety countermeasure based on memory-safe compilation for a realistic CPS environment. **b)** We empirically measure and quantify the tolerability of the induced overhead of the countermeasure based on the real-time constraints of a real industrial control system. **c)** We discuss parameters that affect the absolute overhead 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 experimentation (SWaT) and the memory-safety tools we enforced to our CPS design (ASan and KASan).

2.1 Overview of CPS

CPS constitute complex interactions between entities in physical space and cyber space. Unlike traditional IT systems, these complex interactions are achieved through communication between physical world via sensors and digital world via controllers (PLCs) and other embedded devices. A general architecture of CPS and the interactions among its entities is shown on Figure 2 (in Appendix). Since these systems are real-time, there are latency and reliability constraints. If these real-time constraints are not met, system could run in to an unstable and unsafe state. The devices in a typical CPS are resource constrained too. For example, PLCs and I/O devices have limited memory and computational power. In general, a typical 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 controllers (PLCs).

-*PLCs*: entities that make decisions and issue control commands (based on inputs obtained from sensors) to control plants.

-*Actuators*: entities that implement 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*: a software 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 and alarm history).

2.2 Overview of SWaT

SWaT [28] 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 [29])

Components and specifications. The design of SWaT consists of various components such as real-world PLCs to control the water purification process; a remote input/output (RIO) terminal consisting of digital inputs (DI), digital outputs (DO) and analog inputs (AI); a SCADA system to provide users a local system supervisory and controls; a complex control program written in ladder logic; and so on. It also consists of various system specifications such as a real-time constraints and communication frequencies with other PLCs and the SCADA system. A detailed account ofthe components and specifications is provided in our previous work [30].

2.3 ASan

As discussed on 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] (for the user-space enforcement) as a basis for our empirical study because of its error coverage, high detection accuracy and relatively low runtime overhead when compared to other code-instrumentation based tools. A detailed account on error coverage and runtime overhead of ASan (in comparison with other tools) is provided on [20, 31].

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.

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 4.1, 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.

2.4 KASan

KASan[27, 32] is a fast and dynamic memory error detector tool mainly designed for the Linux kernel. It is also a compile-time code instrumentation memory-safety tool. However, KASan is designed to cover only buffer overflows and dangling pointers not to significantly affect the performance of Linux kernel. Consequently, its runtime overhead is considerably low when compared to ASan. Several kernel memory bugs have been already detected using KASan [33]. Therefore, we chose KASan for the kernel-space enforcement. The current version of KASan is supported only for the x86_64 and ARM64 architectures. Hence, it has compatibility issue with ARM32 architecture, which we have fixed it. As discussed on Section 6.2, the practical tolerability of its overhead (together with ASan) is also evaluated against the real-time constraints of SWaT.

3 Attacker Model and Memory Safety Overhead

In this section, we will introduce our attacker model and formulate its implication in computing memory-safety overheads and its tolerability.

3.1 Attacker model

Memory-safety attacks, such as code injection and code reuse, mainly exploit memory-safety vulnerabilities in the firmware, control software or OS kernel of PLCs. Figure 2 (in Appendix) shows an architectural point of view of memory-safety attacks in CPS. In general, we consider the following five steps involved in a memory-safety attack scenario:

1. Interacting with the victim PLC, e.g., via network connection.
2. Finding a memory-safety vulnerability (e.g. buffer overflow) in the firmware, control software or the OS kernel with the objective of exploiting it.
3. Triggering a memory-safety violation on the PLC, e.g., overflowing a buffer.
4. Overwriting critical addresses of the vulnerable program, e.g., overwriting return address of the PLC program.
5. Using the new return address, diverting control flow of the program to an injected (malicious) code (code injection attacks) or to existing modules of the vulnerable program (code reuse attacks). In the former case, the attacker can get control of the PLC with its injected code. In the latter case, the attacker needs to collect appropriate gadgets from the program, then she will synthesize a shellcode that will allow her to get control of the PLC.

3.2 Modeling Memory Safety Overhead

To ensure memory-safety, firmware and control software of a PLC and kernel of the hosting OS should be compiled with a memory-safety tool. Hence the memory-safety overhead (MSO) will be added to the execution time of the PLC. PLCs handle two main processes – a communication process and a scan cycle

process. The communication process handles any network communication related tasks, e.g., creating connections with communicating entities and receiving and sending network requests. The scan cycle thread handles the main PLC process that involves three operations: scanning inputs, executing the underlying control program and updating outputs. The PLC scan cycle starts by reading the state of all inputs from sensors and storing them to the PLC input buffer. Then, it will execute the control program of the PLC and issue control commands according to the state of sensor inputs. The scan cycle will be concluded by updating output values to the output buffer and sending control commands to the actuators.

The measurement of the actual time elapsed by the PLC scan process, i.e., the time elapsed to scan inputs, execute the PLC program and update outputs is reflected via scan time (T_s). By hardening the PLC with memory-safety protection, we also increase the scan time, which is attributed to the memory safety overhead. Concretely, the memory safety overhead is computed as follows:

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

where $\hat{T_s}$ and T_s are scan time with and without memory-safe compilation, respectively. A detailed account of modeling T_s is provided in our earlier work [30].

3.3 Quantifying Tolerability

A typical CPS involves hard real-time constraints. With memory-safe compilation, we introduce additional overhead, specifically increasing the scan time of a PLC (cf. Equation (1)). We define the notion of tolerability to check whether the induced overhead by the memory-safe compilation still satisfies the real-time constraints imposed by the CPS.

Concretely, a typical scan cycle of the PLC must be completed within the duration of the specified cycle time (T_c). We define two notions of tolerability – 1) for average-case and 2) for the worst-case. In particular, after enabling memory-safe compilation, we compute the scan time (i.e., $\hat{T_s}$) for n different measurements and compute the respective average and worst-case scan time. Formally, we say that the MSO is tolerable in average-case if the following condition is satisfied:

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

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

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

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

4 Enforcing Full-Stack Memory-Safety

It is often mistakenly believed that there is no operating system in PLCs. Most PLCs today are just user-mode applications running on top of POSIX-like operating systems such as Linux OS. For example, Allen-Bradley PLC5 has

Microware OS-9 [34]; Allen-Bradley Controllogix has *VxWorks* [34]; Schneider Quantum has *VxWorks* [34]; Emerson DeltaV has *VxWorks* [34]; LinPAC has *Linux OS* [35]; OpenPLC has *Linux OS* [36]; User-programmable Linux® controllers has *Linux OS* [37]; and so on. Thus, the PLCs work as a software stack running on top of the underlying OS. Therefore, the overall architecture of the control system consists of two main parts: the application stack (that includes the PLC firmware and control software) and the underlying OS.

As discussed in the introduction, the PLC firmware and the control software might have memory-safety vulnerabilities as they are often written in C/C++ due to performance reasons. As such, memory-safety attacks could exploit such vulnerabilities to attack PLCs. Similarly, operating systems are also often implemented in C/C++, hence they might also have memory-safety vulnerabilities. For example, a VxWorks vulnerability (reported on US-CERT [38]) affected Rockwell and Siemens products. Therefore, memory-safety attacks could also exploit vulnerabilities on the operating systems. In particular, attacks could exceptionally target vulnerabilities in the kernel (as also recent trends show in CVE [13]); because the kernel is the core of the machine's OS that is responsible for several critical tasks, e.g. memory management, CPU allocation, system calls, input/output handling, and so on.

To address these security concerns, we proposed a full-stack memory-safety solution that comprises a *user-space* and *kernel-space* memory-safety enforcements. The former refers a memory-safety enforcement to the PLC firmware and control software whereas the later refers a memory-safety enforcement to the OS kernel where the PLC is running on. In this research work, we use OpenPLC controller [36] – a software stack running on top of Linux OS – and the following sections discuss how we enforced the two memory-safety solutions.

4.1 Enforcing User-Space Memory-Safety

As stated on the introduction, our approach to counter memory-safety attacks at user-space level is by secure compiling of the PLCs' firmware and control software. 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 the 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. Therefore, we compiled our OpenPLC firmware ad control software using GCC with ASan enabled.

4.2 Enforcing Kernel-Space Memory-Safety

As discussed on Section 2.4, KASan [27] is a memory-safety tool designed for the Linux kernel. Therefore, we compiled the Raspberry PI Linux kernel (where our controller is running on) with KASan to detect kernel-level memory-safety

violations, such as buffer overflows and dangling pointers. To do so, we configure the kernel with a `KASAN=y` configuration option. But, doing so was not also a straightforward task because of an architectural comparability problem to work on a 32-bit Raspbian kernel. Because KASan is designed only for the x86-64 and ARM64 architectures. To solve the problem, we did a custom kernel build by cross-compiling with a 64-bit Linux OS.

4.3 Detection and mitigation

As discussed on Section 2.3 and 2.4, ASan and KASan instrument 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 a 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 the ASan enforcement, we detected two global buffer overflow vulnerabilities on the OpenPLC Modbus implementation.

The mitigation approach of ASan and KASan 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 exploit the vulnerabilities. However, this approach might not be acceptable in a CPS environment since it highly affects availability of the system and leaves the control system in an unsafe state. Thus, we are currently working on a different mitigation approach to address these limitations.

5 Experimental Design

Unfortunately, SWaT is based on closed-source proprietary Allen Bradely PLCs, hence we cannot modify their firmware to enforce memory-safety solutions. Thus, we designed open-SWaT – a mini CPS based on open source PLCs that mimics features and behaviors of SWaT. By doing so, we managed to conduct our experiment on realistic and closed-source proprietary PLCs, indirectly. We discussed design details of open-SWaT in the following sections.

5.1 open-SWaT

open-SWaT is designed using OpenPLC [36] – an open source PLC for industrial control systems. With open-SWaT, we reproduce operational details of SWaT; in particular we reproduce the main factors (mentioned on Section 6.4) that have significant impact on the scan time and MSO. In general, the design of open-SWaT consists of the following details.

PLCs: we designed the PLCs using 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. It consists of 86 I/O pins that can be directly connected to the I/O devices. 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)).

PLC program: we have designed a control program written in ladder diagram that has similar complexity to the one in SWaT (a sample diagram is shown on Figure 3 (in Appendix)). 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 (normally-open (NO), normally-closed (NC)), and coils (normally-open (NO), normally-closed (NC)). We stated complexity of the program both in terms of number of instructions and lines of code (LOC). The overall PLC program consists of 129 instructions; details are shown on Table 4 (in Appendix). Size of the program (when translated to C code) is 508 LOC.

Communication frequency: the communication architecture of open-SWaT (illustrated on Figure 1) consists of analogous communicating components with that of SWaT. open-SWaT uses both type of modbus communication protocols – modbus TCP (for Ethernet or wireless communication) and modbus RTU (for serial communication). The communication among PLCs is via modbus TCP or modbus RTU whereas the communication between PLCs and the SCADA system is via modbus TCP. Frequency of communication among PLCs and the SCADA system is similar to that in SWaT. The communication between PLCs and Arduino is via USB serial communication. The frequency of receiving inputs from Arduino or sending outputs to Arduino is 100Hz.

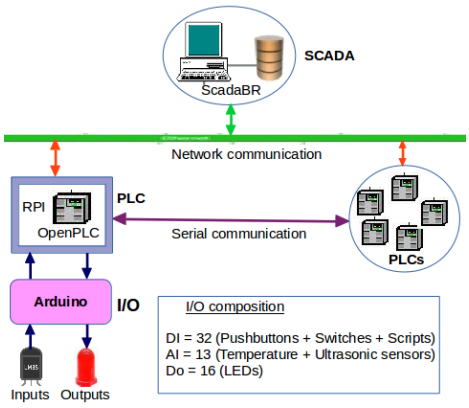

Fig. 1. Architecture of open-SWaT [30]

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

SCADA system: we use ScadaBR [39], a full SCADA system consisting of web-based HMI.

In summary, the design of open-SWaT is expected to be very close to SWaT. In particular, the PLCs (in both cases) are expected to operate similarly. Because their hardware specifications, the inputs they receive from sensors, the PLC program they execute, the control command they issue, the number of nodes they are communicating with, the frequency of communications, and so on, are designed to be similar. Thus, we 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, we plan to replace the PLCs at SWaT with the open-source and memory-safety enabled PLCs of open-SWaT.

5.2 Measurement details

We have implemented a function using POSIX clocks (in nanosecond resolution) that measures execution time of each operation in the PLC scan cycle. The function measures elapsed time of each operation. Results will be then exported to external files for further manipulation, e.g., computing MSO and plotting graphs. We run 50000 scan cycles for each PLC operation to measure the overall performance of the PLC.

6 Evaluation and Discussion of the Results

In this section, we performed a detailed evaluation and discussion of the experimental results to figure out whether the memory-safety tools are 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 and KASan, *performance (efficiency)* – tolerability of its runtime overhead in CPS, and *memory usage* overheads.

6.1 Security

As a sanity check on our configuration, we have evaluated our setup against a wide-range of memory-safety vulnerabilities to explore the detection accuracy of ASan and KASan. The results show that, as in the original paper [20], ASan detects memory-safety violations with high accuracy – without false positives for all the vulnerabilities listed on Table 3 (in Appendix) and rare false negatives for global buffer overflow and use-after-free vulnerabilities due to the exceptions discussed on [20].

As discussed on Section 2.4, KASan's error coverage is purposely limited to buffer overflows and use-after-free vulnerabilities for performance reason. We evaluated its detection accuracy against these vulnerabilities in the Linux kernel and it accurately detects them; no false positives or negatives were discovered or reported so far. Both tools also effectively mitigate the detected violations regardless of the mitigation limitations discussed on Section 4.3.

6.2 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. With our full-stack memory-safety enforcement, i.e., ASan + KASan, the average overhead is 94.32%. The overall performance report of the PLC including the execution time of each operation and its respective MSO is depicted on Table 1.

To evaluate tolerability of this overhead, we have checked if it satisfies the conditions defined on Eq. (2) (for average-case) and Eq. (3) (for worst-case). As shown on Table 1, $mean(\hat{T}_s) = 865.10$µs, and $T_c = 10000$µs. Therefore, according to Eq. (2), the overhead is tolerable for SWaT with the average-case scenario.

To evaluate the tolerability in the worst-case scenario, we check if it satisfies Eq. (3). As shown on Table 1, $max(\hat{T}_s) = 5238.46$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 can conclude that SWaT would tolerate the overhead caused by memory-safe compilation, while significantly increasing its security.

Table 1. Memory-safety overheads (MSO)

Operations	Number of cycles	Network devices	CPU speed (in MHz)	T_s (in µs)		$\hat{T}_s$ (in µs)		MSO (mean)	
				Mean	Max	Mean	Max	in µs	in %
Input scan	50000	6	200	114.94	995.10	204.53	1202.28	89.59	77.95
Program execution	50000	6	200	150.32	716.62	305.59	1982.57	155.27	103.29
Output update	50000	6	200	179.93	1020.47	354.98	2053.61	175.05	97.29
Full scan time	50000	6	200	445.19	2732.19	865.10	5238.46	419.91	94.32

6.3 Memory usage

We also evaluated memory usage overheads of our security measure. Table 2 (in Appendix) summarizes the increase in virtual memory usage, real memory usage, binary size and shared library usage collected by reading VmPeak, VmRSS, VmExe and VmLib fields, respectively, from */proc/self/status*. It shows a huge increase in virtual memory usage (30.45×). This is mainly because of the allocation of large redzones with *malloc*. However, the real memory usage overhead is only 1.40×. These overheads are still acceptable since most PLCs nowadays come with at least 1GB memory size.

6.4 Validation and sensitivity analysis

More generally, how can we evaluate a system's tolerability to overheads? On the one hand, we may perform an empirical analysis such as the one discussed in the previous subsections. But we may also attempt to isolate the individual factors impacting performance on a CPS in order to perform a design-time analysis.

Empirical analysis Suppose the tolerability argument is represented by Φ, where Φ represents Eq. (2) (for average-case) and Eq. (3) (for worst-case). We have empirically measured the scan time of each 50000 scan cycles, say $\hat{T}_{s,1}, ..., \hat{T}_{s,50000}$. Because of the fact that the bar between the worst-case scan time measured, i.e., $max(\hat{T}_s) = 5238.46$µs and the tolerability limit, i.e., $T_c = 10000$µs is still 47.62%, we can fairly conclude that the probability of getting $\hat{T}_s{}'$ such that $\hat{T}_s{}' \not\models \Phi$ is very rare. Therefore, the empirical analysis can be used

as one way of validating tolerability of the MSO even though it cannot prove *completeness* of the argument. However, a more thorough analysis is needed to conclude that there are no corner cases that might suddenly occur and cause more significant delays.

WCST analysis Thus, a deeper analysis to validate the tolerability argument is needed. This is a theoretical analysis (beyond the empirical results) to show that there would not occur a new WCST $\hat{T_s}'$ such that $\hat{T_s}' > T_c$. For simplicity, let us refer the occurrence of the condition $\hat{T_s}' > T_c$ as an *intolerability* condition. For this analysis, first we experimentally identified the main factors that can have significant effect on the PLC scan time and MSO. We discussed below how the factors can affect the scan time and why they would not lead to the *intolerability* condition.

-CPU speed ***($S_{CPU} \in \mathbb{R}$)***: obviously, clock speed of the processor is a major factor for the PLC performance. It determines how much clock cycles the CPU performs per second, hence it determines how much instructions the PLC can process per second. S_{CPU} affects all operations of the PLC. However, since S_{CPU} is fixed with "userspace" governor, it would not lead to *intolerability*.

Memory size ***($S_M \in \mathbb{R}$)***: size of memory is fixed. The memory size needed for memory mapping and redzones allocation (due to the memory-safe compilation) is already allocated at compile-time. Cache memory size is not also a big issue in CPS. Because CPS data such as sensor data, control commands and state information get updated very frequently. Thus, data caching is not that much relevant in CPS. Therefore, S_M would not lead to *intolerability*.

-Number of sensors ***($N_S \in \mathbb{N}$)***: the number of input devices (sensors) connected to the PLC is one factor that significantly affect the PLC scan time and MSO. Because, the time to scan inputs depends on the number of sensors connected with the PLC. However, N_S is fixed, hence it would not cause the *intolerability* condition to happen.

-Number of actuators ***($N_A \in \mathbb{N}$)***: the number of output devices (actuators) connected to the PLC is also another factor that has significant effect on the PLC scan time and MSO. Because, the time to update outputs depends on the number of output devices connected with the PLC. However, since N_A is fixed, it would not lead to *intolerability*.

-Complexity of the PLC program ***($C_P \in \mathbb{R}^Z$)***: As discussed on Section 5.1, the PLC program can consist of various types of instructions. Each instruction has its own execution time. Therefore, C_P can be expressed in terms of the number and type of instructions that the overall program consists of (Z = {number of instructions, type of instructions}). As such, it is a major factor for the PLC scan time as it affects the control program execution time. However, C_P is fixed and the program does not also contain loops or recursion functions. Thus, it would not lead to the *intolerability* condition.

-Communication frequency ***($C_F \in \mathbb{R}$)***: the PLC communicates with various devices such as RIO (sensors and actuators), other PLCs and SCADA systems. The communication frequency can be expressed in terms of the number of pack-

ets the PLC sends or receives to/from other communicating devices. Handling all such communications can take significant amount of time. In particular, it significantly affects the PLC's performance when the PLC handles the concurrency issues between the scan cycle and communication threads to access shared resources, such as shared buffers [30]. Therefore, the communication frequency between the PLC and other communicating entities is another factor for the PLC scan time. However, when the PLC communicates with n nodes, it receives or sends packets with a constant rate. Thus, the C_F is fixed. In addition, realistic PLCs (as real-time systems) efficiently handle concurrency problems. Therefore, the C_F would not result the *intolerability* condition.

We also performed a sensitivity analysis on the factors in regard to its effect on the PLC scan time and MSO. This analysis will help us to extrapolate mathematical formulas predicting the expected MSO and its tolerability to a given CPS. A detailed account of our sensitivity analysis is provided in our previous work [30].

7 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 overheads in the CPS environment.

In our earlier work [30], we enforced ASan to a CPS environment and measured its runtime overhead (81.82%). However, it was only a user-space enforcement and the critical kernel-level security concern was ignored. To address that limitation, we enforced a full-stack memory-safety, i.e., ASan + KASan, in our current work. With a similar setup but a different kernel configuration, the average overhead of the proposed solution is 94.32%. Meaning, it incurs an additional overhead of 12.5%, but with a significant boost in security. To enhance comprehensiveness of our experimental results, we also increased the number of scan cycles (whose scan time is empirically measured) from 10000 to 50000.

SoftBoundCETS is a compile-time code-instrumentation tool that detects all violations of spatial memory-safety (SoftBound [21]) and temporal memory-safety (CETS [22]) in C. It is a complete memory-safety tool that works under the LLVM environment. 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. [40] enforced efficient memory-safety solution for TinyOS applications by integrating Deputy [41], an annotation based type and memory-safety compiler, with nesC [42], 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 elimina-

tion. However, their solution has limitations to apply it in a CPS environment, because it is dependent on runtime libraries of TinyOS.

Zhang et al. [43] modeled the trade-off between privacy and performance in CPS. While he leveraged the differential privacy approach to preserve privacy of CPS, he also analyzed and modeled its performance overhead. He 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 it is not from memory-safety perspective.

Stefanov et al. [44] proposed a new model and platform for the SCADA system of an integrated CPS. With the proposed platform, he modeled real-time supervision of CPS, performance of CPS based on communication latencies, and also he assessed and modeled 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 memory-safety overheads. Vuong et al. [45] tried to evaluate performance overhead of a cyber-physical intrusion detection technique. But, it was not on memory-safety either.

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 [46]). 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 that enforced a full-stack memory-safety solution specifically to the CPS environment, and that measured and evaluated tolerability of the induced memory-safety overhead in accordance to the real-time constraints of cyber-physical systems.

8 Conclusion

In this work, we presented the results of implementing a strong full-stack memory-safety enforcement in a simulated albeit realistic industrial control system using ASan and KASan. 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 constraints of SWaT can be largely met even when implementing a strong memory-safety countermeasure 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 and urban transportation systems. 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. LeeSeshia.org (2015)
3. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: ISORC'08
4. MITRE: Common Vulnerabilities and Exposures. https://cve.mitre.org/
5. CVE-5814. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5814
6. CVE-6438. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6438
7. CVE-6436. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-6436
8. CVE-0674. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0674
9. CVE-1449. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1449
10. CVE-0929. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0929
11. CVE-7937. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7937
12. CVE-5007. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5007
13. (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)
14. Berger, E.D., Zorn, B.G.: Diehard: Probabilistic memory safety for unsafe languages. In: PLDI'06. (2006)
15. Novark, G., Berger, E.D.: Dieharder: Securing the heap. In: CCS'10. (2010)
16. Kharbutli, M., Jiang, X., Solihin, Y., Venkataramani, G., Prvulovic, M.: Comprehensively and efficiently protecting the heap. In: ASPLOS'06. (2006)
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++. In: GCC Developer's Summit, Red Hat Inc. (2003)
27. KASAN: The Kernel Address Sanitizer. https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html (2018)
28. iTrust: Secure Water Treatment (SWaT) testbed. https://itrust.sutd.edu.sg/research/testbeds/secure-water-treatment-swat/
29. 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)

30. 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. Lecture Notes in Computer Science. Volume 10683., Springer International Publishing (2017) 127–144
31. AddressSanitizer github repository. https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools
32. KASAN Wiki: The Kernel Address Sanitizer Wiki. https://github.com/google/kasan/wiki (2018)
33. KASAN Bug Report: List of kernel bugs detected by KASan. https://github.com/google/kasan/wiki/Found-Bugs (2018)
34. TOFINO SECURITY. https://www.tofinosecurity.com/blog/plc-security-risk-controller-operating-systems/
35. LinPAC. http://www.icpdas.com/root/product/solutions/pac/linpac/linpac-8000_introduction.html/
36. OpenPLC. http://www.openplcproject.com/
37. WAGO: Linux Programmable Fieldbus Controller
38. CERT.ORG: Vulnerability Notes Database
39. ScadaBR. http://www.scadabr.com.br/
40. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: SenSys'07. (2007) 205–218
41. The Deputy project. http://deputy.cs.berkeley.edu (2007)
42. 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)
43. Zhang, H., Shu, Y., Cheng, P., Chen, J.: Privacy and performance trade-off in cyber-physical systems. IEEE Network **30**(2) (2016) 62–66
44. 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
45. Vuong, T.P., Loukas, G., Gan, D.: Performance evaluation of cyber-physical intrusion detection on a robotic vehicle. In: IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing. (2015)
46. 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)

Appendix

Table 2. Memory usage overheads (in MB)

Category	Original	Instrumented	Increase
Virtual memory usage	20.412	621.580	30.45×
Real memory usage	8.172	11.476	1.40×
Binary size	0.138	0.316	2.29×
Shared library usage	2.832	4.300	1.52×

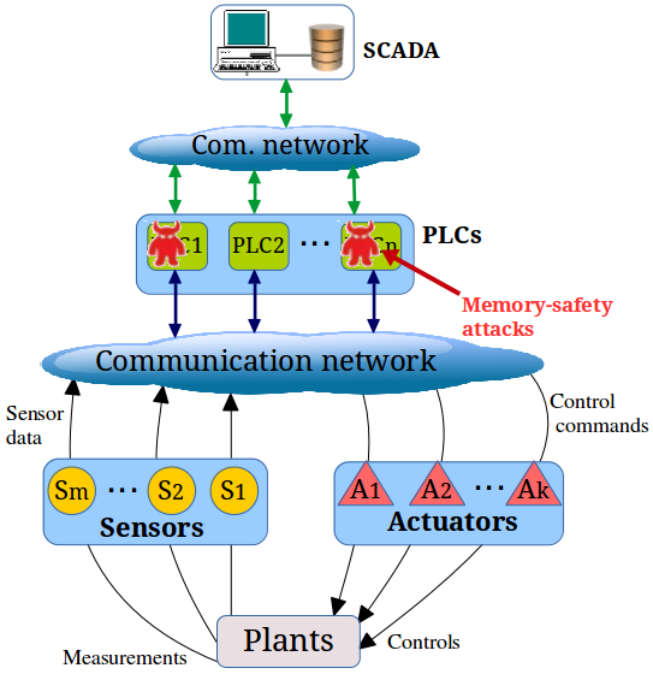

Fig. 2. The CPS architecture and memory-safety attacks [30]

Table 3. Detection accuracy of ASan

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

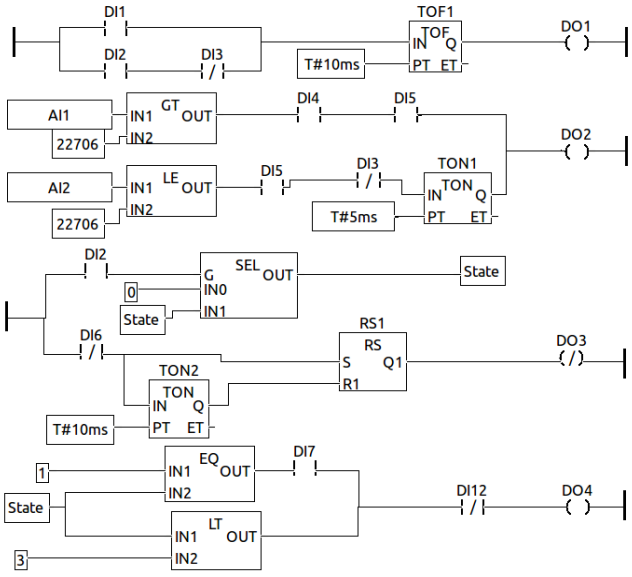

Fig. 3. Sample PLC program in ladder diagram [30]

Table 4. 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