Bespoke Security for Resource Constrained Cyber-Physical Systems

Miguel A. Arroyo

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Bespoke Security for Resource Constrained Cyber-Physical Systems

Miguel A. Arroyo

Cyber-Physical Systems (CPSs) are critical to many aspects of our daily lives. Autonomous cars, life saving medical devices, drones for package delivery, and robots for manufacturing are all prime examples of CPSs. The dual cyber/physical operating nature and highly integrated feedback control loops of CPSs means that they inherit security problems from traditional computing systems (e.g., software vulnerabilities, hardware side-channels) and physical systems (e.g., theft, tampering), while additionally introducing challenges of their own. The challenges to achieving security for CPSs stem not only from the interaction of the cyber and physical domains, but from the additional pressures of resource constraints imposed due to cost, limited energy budgets, and real-time nature of workloads. Due to the tight resource constraints of CPSs, there is often little headroom to devote for security. Thus, there is a need for low overhead deployable solutions to harden resource constrained CPSs. This dissertation shows that ***security can be effectively integrated into resource constrained cyber-physical system devices by leveraging fundamental physical properties, & tailoring and extending age-old abstractions in computing***.

To provide context on the state of security for CPSs, this document begins with the development of a unifying framework that can be used to identify threats and opportunities for enforcing security policies while providing a systematic survey of the field. This dissertation characterizes the properties of CPSs and typical components (e.g., sensors, actuators, computing devices) in addition to the software commonly used. We discuss available security primitives and their limitations for both hardware and software. In particular, we focus on software security threats targeting memory safety. The rest of the thesis focuses on the design and implementation of novel, deployable approaches to combat memory safety on resource constrained devices used by CPSs (e.g., 32-bit processors and microcontrollers).

We first discuss how cyber-physical system properties such as inertia and feedback can be used to harden software efficiently with minimal modification to both hardware and software. We develop the framework You Only Live Once (YOLO) that proactively resets a device and restores it from a secure verified snapshot. YOLO relies on inertia, to tolerate periods of resets, and on feedback to rebuild state when recovering from a snapshot. YOLO is built upon a theoretical model that is used to determine safe operating parameters to aid

a system designer in deployment. We evaluate YOLO in simulation and two real-world CPSs, an engine and drone.

Second, we explore how rethinking of core computing concepts can lead to new fundamental abstractions that can efficiently hide performance overheads usually associated with hardening software against memory safety issues. To this end, we present two techniques: (i) The Phantom Address Space (PAS) is a new architectural concept that can be used to improve N-version systems by (almost) eliminating the overheads associated with handling replicated execution. Specifically, PAS can be used to provide an efficient implementation of a diversification concept known as execution path randomization aimed at thwarting code-reuse attacks. The goal of execution path randomization is to frequently switch between two distinct program variants forcing the attacker to gamble on which code to reuse. (ii) Cache Line Formats (Califorms) introduces a novel method to efficiently store memory in caches. Califorms makes the novel insight that dead spaces in program data due to its memory layout can be used to efficiently implement the concept of memory blacklisting, which prohibits a program from accessing certain memory regions based on program semantics. Califorms not only consumes less memory than prior approaches, but can provide byte-granular protection while limiting the scope of its hardware changes to caches. While both PAS and Califorms were originally designed to target resource constrained devices, its worth noting that they are widely applicable and can efficiently scale up to mobile, desktop, and server class processors.

As CPSs continue to proliferate and become integrated in more critical infrastructure, security is an increasing concern. However, security will undoubtedly always play second fiddle to financial concerns that affect business bottom lines. Thus, it is important that there be easily deployable, low-overhead solutions that can scale from the most constrained of devices to more featureful systems for future migration. This dissertation is one step towards the goal of providing inexpensive mechanisms to ensure the security of cyber-physical system software.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to especially thank my advisor Prof. Simha Sethumadhavan for his encouragement that kept pushing me onward in this long journey even when I doubted myself. To put simply, without his support I would not have come as far as I have today. He has not only shaped my research personality, but has passed on his ability to have faith in a vision.

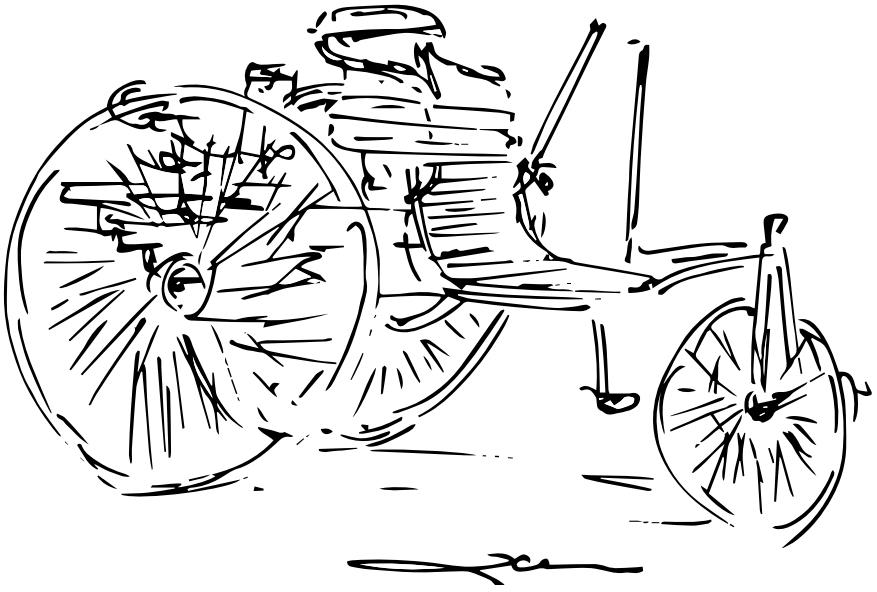
I am also very grateful to the support and effort of those who have helped me in my journey. Chief among my collaborators is Mohamed Tarek Ibn Ziad with whom I have shared countless hours in long brainstorming sessions that have led to many of the contributions presented here. On YOLO, he contributed greatly to the development of the theoretical control framework. On PAS, Mohamed took the lead on the hardware design, while I developed the compiler modifications and experimental setup. On SPAM, our long debugging sessions together were instrumental to our accomplishments. Not only did we work more efficiently to discover bugs, but it also forced us to reason about SPAM in new ways. Ultimately, this led to key performance optimizations where Mohamed took the lead, while I developed a source-to-source compiler pass to provide additional security benefits.

Additionally, I would like to thank the wonderful cast of collaborators whose efforts have made all of these contributions possible: Dr. Hiroshi Sasaki, Dr. Kanad Sinha, and Koustubha Bhat with whom I worked with on Califorms to define the dead-bytes concept, security guarantees, and compiler implementation; Dr. Hidenori Kobayashi with whom I worked countless days getting our hands dirty in the basement of Mudd on an engine and crashing our drone in the park while developing YOLO; Evgeny Manzhosov who contributed to the hardware implementation of PAS and meticulously handled experimental data collection; Finally, Prof. Vasileios P. Kemerlis and Prof. Junfeng Yang for their discussions on PAS and YOLO, respectively.

To my wife, Zihong Su

Part I

Cyber-Physical Systems



1 Introduction

Over the past century, many purely mechanical devices have evolved to become computer controlled. The automobile, for example, has transitioned from being a "horseless carriage" to today's near autonomous versions entirely reliant on computers. Automobiles are just one of many instances of cyber-physical systems. Life saving medical devices, drones for package delivery, and robots for manufacturing are all prime examples. Cyber-physical systems (CPSs) play an increasingly important role in our everyday lives, changing the way we continuously interact with the world.

The term cyber-physical system (CPS) emerged just over a decade ago as an attempt to unify problems that span across multiple research disciplines [1]. Independently, these research disciplines have rich histories ranging from control theory in 1868, cybernetics feedback in 1949, embedded systems in 1961, software engineering in 1968, to ubiquitous computing in 1988. The convergence of technological advances, including the miniaturization of sensors, batteries, and energy efficient processors has led to the rise of rich ecosystems of systems capable of controlling increasingly complex physical processes [2, 3].

As the proliferation of CPSs increases, so do the security risks. The consequences of unintentional faults or malicious attacks could have a severe impact on human lives and the environment. Thus, proactive and coordinated efforts are needed to strengthen the security and reliance of cyber-physical systems.

1.1 What are Cyber-Physical Systems?

Cyber-Physical Systems are defined by their hybrid nature involving the cyber (digital) and physical (analog) domains. CPSs are a collection of individual task-oriented systems whose interactions provide functionality that is more than that which can be achieved by any one individual; in other words, CPSs exhibit *emergent* behavior. For example, consider the components that make up a car: an engine, wheels, chassis, etc. None

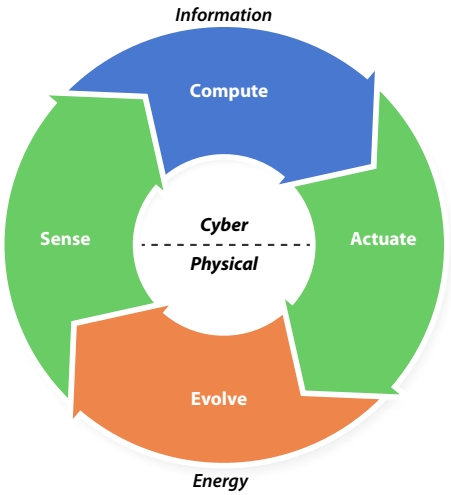


FIGURE 1.1: A conceptual CPS model.

of the individual components of a car can be driven, only when they are combined together does it become possible to go from point A to B.

Cyber-Physical Systems are composed of physical process, sensors, actuators, and computational units. A conceptual model of how a CPS operates is shown in Figure 1.1. The upper half of the figure represents the cyber domain, where information is exchanged, and the lower half represents the physical domain, where energy is transferred. Sensors and actuators lie at the boundary between these domains converting energy into information and vice-versa, respectively. Most importantly, CPSs operate in a feedback loop between the cyber and physical domains. In other words, the sensed information is used to compute an actuation action that causes a physical process to evolve, which is then sensed and so on and so forth. This feedback loop is used to try and ensure the physical process's stability and safety. A more technical discussion of these concepts is covered in Chapter 3.

COMPARISON WITH INTERNET OF THINGS Despite their distinct origin [4], the Internet of Things (IoT) and CPS refer to a related set of trends integrating physical devices with computational components to enhance performance and functionality. In this thesis, we clarify and extend the relationship between CPSs and IoT though the perspective of a unified framework [5]. CPSs and IoT are usually considered separate fields with their own publications and communities. In this thesis, we view IoT as a subset of CPSs, where there is no feedback loop between the cyber and physical domains. In other words, IoT do not have an actuation component. We discuss the distinction between CPSs and IoT in more detail in § 2.3.

1.2 How is Security different from Safety?

Ensuring safety has long been among the most important design goals of CPS. Safety is aimed at protecting systems from accidental failures, while security is focused on protecting systems from intentional (i.e., adversarially induced) faults. They share identical goals, however, without security we argue there can be no safety. One can reason about it in the following way, the lock on your door helps to ensure that your belongings are not stolen. The lock is responsible for maintaining the safeguards you expect in order to ensure safety. In this example, it can prevent your fire-extinguisher, which is meant to keep you safe in the event of a fire, from being stolen. More generally, failure to perceive all errors in a system (for safety) is not fatal; a system built with enough tolerance could be designed to withstand errors. In contrast, failure to perceive errors (for security) can often be fatal.

Traditionally, safety and security have been handled by very distinct groups of people. As a result, safety-oriented design methodologies tend to not fully cover cyber-derived threats [6]. As we will present in this dissertation, in the face of the emerging threats targeting CPSs, considering these two independently is no longer possible.

1.3 How is Cyber-Physical System Security Different?

Individually, the security of the components of a CPS have been well studied. The cyber subsystem faces many of the same threats that plague IT security (e.g., confidentiality, integrity, availability), and the physical subsystem faces many of the same threats that previous mechanical incarnations faced. However, reasoning about the effects of the compositionality of components is challenging as security is a global property [7, 8, 9]. Complex interactions between the cyber and physical domains introduce many new challenges and threats. These interactions make reasoning about security more difficult than considering the constituent components in isolation; that is, emergent behaviors can lead to new threats. For instance, in the case of a car, its ability to move introduces the need to consider collisions as a potential threat vector.

1.3.1 Software Security

Electronics and the corresponding software that drive CPS make up a large fraction of the total system both in terms of components and cost. As a result, the importance of software security for these systems cannot

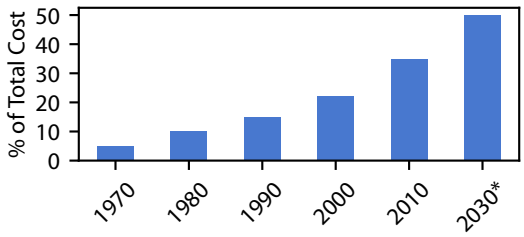


FIGURE 1.2: Automotive electronics cost as a share of total car cost from 1950-2030 [10].

be overstated. To put this into perspective, let us consider the automotive industry. Most of the automotive innovations taking place today arise from electronics rather than mechanics. The cost contribution of automotive electronics increased from around 20% in 2007 to about 40% in 2017 [10]. The cost of electronics in automobiles will continue to grow with the advances in electric vehicles and autonomous driving and is projected to account for 50% by 2030 as shown in Figure 1.2.

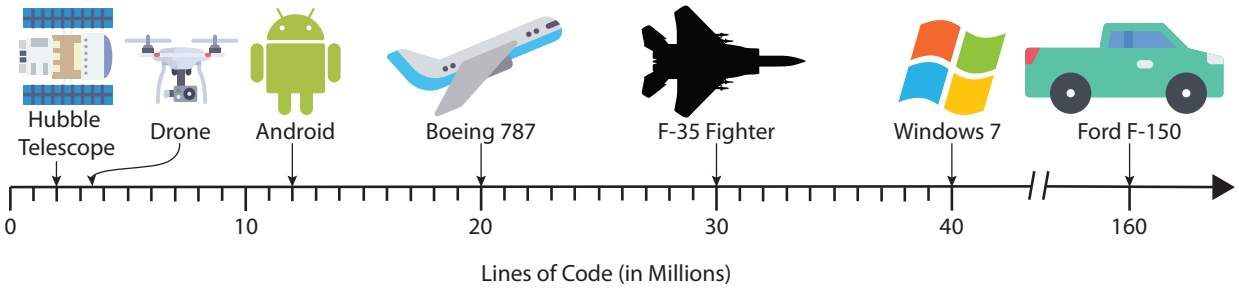


FIGURE 1.3: Estimated software complexity in lines of code [11].

As more of a CPS becomes computerized, it is natural for the complexity of the software necessary to run these systems to increase as is evident by the data shown in Figure 1.3. It is relevant to note that a large portion of CPS software is as complex (in terms of lines of code) as large general-purpose (GP) software — the automotive industry is an order of magnitude more complex. Defects in software are an inevitability at some point, in particular for large, complex projects. According to previous research, the industry average defect rate is around 1-25 bugs for every 1,000 lines of code [12]. Thus, as more of a CPS relies on software, the higher the likelihood of security vulnerabilities.

1.3.1.1 Embedded Software Challenges

The embedded devices found in CPSs impose many challenges that transcend the ordinary requirements of GP software. These challenges can be broadly categorized as follows:

- Real Time. CPSs pose additional constraints that must be addressed in "real-time" due to their reactive behavior. The cyber components must provide an expected control action within a maximum specified time or deadline. The level of real-time is classified according to the utility of a result after a deadline. If a result has utility after the deadline it is classified to be *soft* real-time. Otherwise, if a result has zero utility after the deadline and missing it may prove to be catastrophic to the system, it is classified to be *hard* real-time.

- Availability. CPS software may operate for decades without service and with limited possibility for upgrades or patches [13]. While the possibility for over-the-air (OTA) updates is seeing more widespread adoption, they bring challenges of their own [14].

- Heterogeneity. The embedded world is heavily fragmented. A wide variety of processors, sensors, and hardware parts exist [15]. While there has been much consolidation in the processor instruction set used (i.e., the industry has mostly centered around ARM which according to a recent survey accounts for 67% [16]), vendors implement their own custom features to extend basic architectures and create product differentiation. Sensors and various special purpose hardware present the biggest challenges which due to the incredible diversity, makes device drivers difficult to develop. All these factors lead to software that is often written for the lowest common denominator in terms of capabilities. As a result, both hardware and software often lack the features found in the more standardized GP space.

- Testability. By definition, the embedded systems found in CPS interface with the physical world in some way. Physical interactions make it difficult to properly test software at scale due to cost and/or physical limitations. Simulators and emulators of physical processes provide a partial solution, but come at a cost to fidelity that may not capture security threats.

- Limited Resources. Embedded systems by nature only perform a few specific actions and therefore do not need the computing power of GP computers. All resources considered non-essential to their task are eliminated to reduce power consumption, space, and ultimately cost [17, 18]. Thus, embedded

software is typically constrained by smaller memory, limited data-processing capabilities, and stripped down hardware functionality.

1.3.1.2 Memory Safety

Much like GP software, 80% of embedded software is predominantly written in C/C++ due to strict performance and space requirements [15]. These languages suffer from various memory (un)safety issues that are the root-cause of many software security exploits. Historically, program memory safety violations have provided a significant opportunity for exploitations: for instance, a recent report from Microsoft revealed that the root cause of more than half of all exploits targeting their products were software memory safety violations [19]. Similarly, the Project Zero team at Google, reports that memory safety issues are the root-cause of more than 80% of listed CVEs for 0-day vulnerabilities seen from 2014-2019 [20]. As a result of CPS software's heavy use of unsafe languages, memory safety vulnerabilities are as much a threat for CPS as they are for GP systems.

The challenges of embedded devices present additional difficulties compared to the GP space when it comes to designing and implementing memory safety mitigations. Many of the features that are necessary for defensive techniques developed for GP devices may not exist in the embedded space whether it be due to limited resources, timing constraints, or heterogeneity of hardware. Due to a CPS's unique ability to exert physical actions, the need for providing memory safety is just as strong, if not stronger, than in the GP computing world.

1.3.2 Physical Security

A CPS does not only have to contend with cyber attacks, but also those which manipulate sensors and actuators in the physical domain. CPS rely on the trustworthiness of sensors and actuators to interface between the physical and digital domains. Although information security methods such as cryptography provide a solution for secure communication between parties in the digital domain, the equivalent for the physical environment has yet to be developed. This disconnect between the environment, the method of measurement, and the analog-to-digital (or digital-to-analog) conversion of measurement signals can be exploited by adversaries to manipulate control actions. For instance, malicious acoustic interference can influence the output of sensors trusted by software in systems ranging from medical devices [21], drones [22], to autonomous

vehicles [23] in order to hijack control. Moreover, actuators can be used to exfiltrate digital data through the analog domain from isolated networks [24, 25].

Billions of deployed sensors and actuators lack protections against these threats and researchers have repeatedly shown how an adversary can control their outputs with malicious signals [21, 22, 26, 27, 28]. Protecting against these attacks is difficult because the vulnerabilities lurk deep within the underlying physics of their implementation. While these attacks are powerful, the physical limitations they impose may make them much less practical than the low-hanging cyber threats that plague the embedded devices controlling a CPS.

1.3.3 Why has security lagged behind?

In addition to the technical challenges discussed above, a number of other factors have led CPS security to lag behind the mature GP computing space. These factors include:

- Misaligned Incentives. As a result of economic factors, manufacturers are likely to under-invest in security measures to keep costs low [29, 30]. There is usually little incentive on part of individual vendors and manufacturers that supply the many components of a CPS to provide security features that can be leveraged by others along the supply chain. It is easy to see how this can lead to misaligned incentives between the relevant parties at every link in the supply chain.

- Obscurity into inner-workings. To protect their products, many vendors either obfuscate their designs or encrypt the relevant software to prevent an understanding of the internals. Unfortunately, this comes with additional side-effects that in the case of CPS are more pronounced than in other domains. Without access to the internals and software that make up a CPS's constituent components, CPSs with long lifespans can be hard hit as vendors abandon support. This means that it is not unusual to find CPSs with old known vulnerabilities that have long since been addressed due to the difficulty of patching. This makes CPSs interesting targets for attackers.

- Regulatory Environment. Many CPSs are required to attain certification to operate in safety-critical applications. Certification serves as a formal assurance that the system has met relevant technical standards designed to ensure it delivers its intended functionality safely and securely. Earning this certification is challenging and costly [31]. Unfortunately, this process serves to deter future modifications to

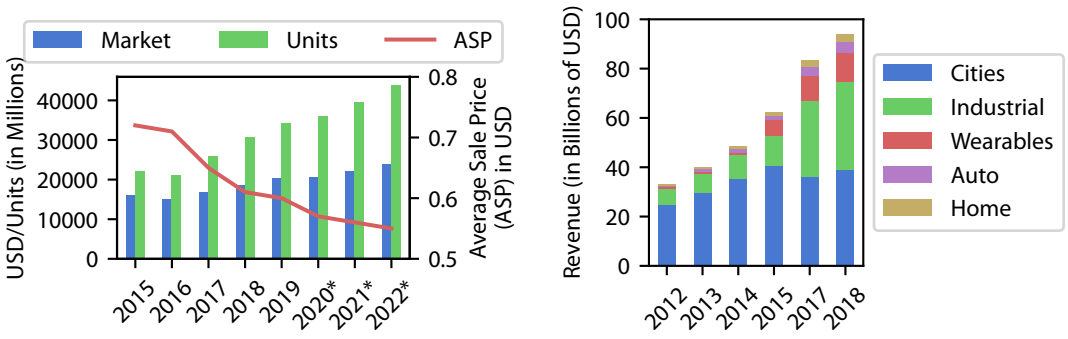
a system, say to patch a security vulnerability [32]. With the tight cost constraints that these systems already face, this additional friction negatively impacts the adoption of security protection mechanisms.

- Lack of Awareness/Adoption. The lack of security awareness for CPSs, until recently, means that systems have not been designed to take many threats into consideration. Even perfectly secure hardware and software may be compromised if interactions between components is not accounted for. Current sensors are not generally designed with malicious signal injections in mind. Similarly, the control algorithms that rely on these sensors are generally not designed to cope with malicious sensors. Moreover, the cyber components of a CPS were generally assumed to operate in an isolated (air-gapped) environment inaccessible to attackers, often resulting in security to be put aside. Coupled with performance concerns, the lack of awareness of CPS security has greatly impacted the adoption and implementation of security features in many systems.

1.3.3.1 Current & Future Trends

According to a recent survey, security has become the top concern among embedded developers with over 39% of correspondents now aware of the risks [16]. The flurry of attacks in recent years has no doubt helped raise security awareness. Processor manufacturers have also taken note, and have started to release processors with more security features to try and bridge the gap between GP processors and the stripped down versions found in embedded systems [33]. Open source embedded operating systems being developed by hardware manufacturers such as ARM's mbed now provide simpler out-of-the-box support for various security primitives [34]. As more developers flock to these platforms, the adoption of security primitives will hopefully increase resulting in less feature fragmentation. Additionally, companies in the CPS space are trending towards more vertically integrated business models [35]. Vertical integration may ultimately lead to more consolidation and less heterogeneity among devices. Time will tell if vertical integration will help reduce the fragmentation of security features. In addition to the compute side, sensor and actuator manufacturers have also taken note of the new physical threats, issuing important guidelines to system designers on how to mitigate risk [36].

Finally, the cost of components such as processors used by many CPSs (i.e., microcontrollers) are on a rapid decline (see Figure 1.4a). The average sale price (ASP) fell to the lowest point ever in 2017 with pricing expected to continue to drop over the next few years [37]. CPSs across various sectors (i.e., autonomous



(A) Microcontroller market forecast.

(B) CPS growth across various sectors.

FIGURE 1.4: CPS market trends [37].

vehicles, robotic manufacturing, smart-homes, smart-cities, etc as shown in Figure 1.4b) have demonstrated strong growth that is only projected to continue for the foreseeable future [37]. Given these trends, security practitioners will need to seek ways to address the challenges faced by resource constrained devices that will remain in operation for the foreseeable future.

1.4 Contributions

Much of the work for CPS security today focuses on highlighting novel attacks that exploit sensors [38, 39]. While these attacks are important to understand, their utility is bounded by factors such as physical proximity and level of control. Given the current state-of-affairs, software may arguably be the weakest and most flexible threat vector for adversaries. Software attacks offer the ability for remote access and fine-grained control of a system. The proposed thesis is a study in understanding the ways we can strengthen the security of resource constrained cyber-physical system software by leveraging unique physical properties and revisiting fundamental abstractions to make security more efficient.

1.4.1 Leveraging Physical Properties for Software Security

Cyber-Physical Systems provide a unique opportunity to leverage the physical environment to strengthen the security of its cyber components. By leveraging CPS properties, security may be more easily retrofitted into existing systems, or new techniques may be developed at a lower performance cost making them applicable for resource constrained devices.

You Only Live Once (YOLO)

YOLO is among the first solutions that leverages the physical properties of CPS, namely *inertia*, to provide a domain-specific mitigation against software security vulnerabilities. The fundamental property of inertia is the tendency of a system to resist external forces (e.g., motion, temperature, etc). In other words, inertia helps keep an object at equilibrium, providing a natural ability to tolerate transient imperfections during operation.

YOLO is simple to describe. It goes through two operations: Reset and Diversify, as frequently as possible — typically in the order of a few seconds. Due to inertia, a CPS can remain safe even under frequent resets. Resets mitigate attacks that aim to achieve persistence on a device by bounding the time horizon over which an attacker can induce malicious inputs. Diversification is then used to force an adversary to develop new attack strategies over time.

1.4.2 Revisiting Age-old Computing Abstractions for Efficient Security

YOLO aims to discard all program data periodically, relying on the physical environment to recreate this discarded data. In reality, not all program data can be re-observed from the physical environment. For example, configuration and calibration parameters are not recoverable without intervention from an operator. As a consequence, a subset of program data must remain alive for the duration of program execution, perhaps even longer if it is stored in persistent memory. Additionally, short reset periods that provide stronger security may not be feasible for systems with limited inertia. Mechanisms that can ensure security in these longer stretches of time or that can protect data that must be persisted are valuable. Moreover, subsets of CPS, such as IoT devices, do not have the necessary inertia required to implement YOLO. The next two techniques presented in this thesis, PAS and Califorms, provide solutions that can not only be layered with YOLO, but are applicable to a broad spectrum of resource constrained devices[1].

Phantom Address Space (PAS)

Safety and fault tolerance have a long history in the CPS domain. The concept of N-version programming was first introduced in 1977 [40, 41] as an approach to fault-tolerant software and has seen widespread use in many CPSs such as airplanes [42]. Software is considered to be fault-tolerant if it can continue to supply the expected outputs after software errors are triggered. In N-version programming, multiple variants of a

[1] These techniques can also be extended to general-purpose systems.

program are executed. The redundancy is intended to compensate for the software errors as the execution of the multiple variants are compared for inconsistencies. N-version programming has also been extended to improve the security of systems [43] — where it is more commonly referred to as N-variant execution. In this model, an attacker can break the system, if and only if they can affect all instances the same way, and force them to produce the same output. However, the simple yet powerful technique of N-version programming suffers from high memory and runtime performance overheads due to the need of replicating program execution (i.e., code and state) in addition to hardware.

The Phantom Address Space (PAS) is an architectural concept that can be used to improve upon N-version systems by (almost) eliminating the overheads associated with handling replicated execution. The basic idea is to provide different code views using only modifications to memory addressing functions without duplicating code. By doing away with costly resource duplication, PAS is uniquely suited for deployment on low memory CPS devices. We use PAS to provide an efficient implementation of a security diversification concept known as execution path randomization aimed at thwarting code-reuse attacks. The basic idea is to frequently switch between multiple program variants forcing the attacker to gamble on which code to reuse.

Cache Line Formats (Califorms)

Embedded devices are often tailor-made for the specific CPS they are integrated into. To save on cost and energy, these devices are designed to have minimal headroom in terms of processing (i.e., in the 100−400MHz range) and memory (i.e., in the 4KB-2MB range). Moreover, the minimal headroom and time sensitive nature of CPS makes the addition of non-physically critical tasks, like security, challenging. Providing memory safety often requires additional metadata in order to provide access control for memory regions. The way access control metadata is stored and checked is essential to minimizing the memory footprint required to implement memory safety on resource constrained CPSs.

Califorms builds on an idea called memory blacklisting, which prohibits a program from accessing certain memory regions based on program semantics. Califorms makes the novel insight that metadata used for blacklisting can be stored in dead spaces in program memory directly addressing the limited memory availability in CPS devices. Califorms shows how metadata can be integrated into existing micro-architectures by changing the cache line format. The limited scope of the hardware changes makes Califorms agnostic to a given processor architecture making it well suited for the heterogeneous CPS ecosystem. Using these observations, Califorms based systems reduce the performance overheads of memory safety while provid-

ing byte-granular protection with low overheads. Thus, Caliform's granularity and cost make for a practical extension to the coarse-grained memory protection units commonly found in inexpensive processors.

2 A Cyber-Physical System Framework

This chapter highlights the fundamental properties of cyber-physical systems and proposes an ontological framework that can be used to identify and enumerate security vulnerabilities and safeguards.

2.1 The Origins of Cyber-Physical Systems

The term cyber-physical system emerged just over a decade ago as an attempt to unify problems spanning multiple research disciplines [1]. Independently, the research disciplines that make up CPSs have rich histories of their own ranging from: control theory in 1868, cybernetic feedback in 1949, embedded systems in 1961 and beyond that have contributed to our modern day view.

Control theory dates to the 19th century when its theoretical groundwork was laid out by James Clerk Maxwell [44]. Control theory is concerned with the mathematical foundations that ensure *stability* (in an optimal manner) for *dynamical* systems, those which evolve over time. Stability is achieved via feedback, that is when the outputs of a system are used as inputs. The concept of feedback is central to control theory and CPSs in general. As we will discuss in § 3.2, there are two fundamental types of feedback (i.e., open and closed) which today are implemented in software.

The field of cybernetics studies the structure of feedback systems modeled by control theory. It established the foundations for the design of complex systems before the advent of digital computers. Cybernetics as a discipline, formalized by Norbert Wiener in the mid 20th century (circa 1950s) [45], serves as the closest predecessor to today's study of cyber-physical systems. Cybernetics developed a basic framework for discussing feedback, stability, equilibrium, disturbance, entropy, and information to name a few.

By the 1960s and 1970s, feedback control systems evolved with the introduction of digital computers. The development of microprocessors was widely adopted in industrial automation, aircraft, and automobiles. For instance, electronic fly-by-wire systems in airplanes were first used in passenger planes dating back to the

Concorde in 1969 [46], and microprocessor based engine control units (ECUs) were introduced in cars as early as 1968 [47]. Much like how cybernetics, integrated higher-level abstractions to control theory, CPSs aim to integrate concepts of software development into the study of physical systems. In CPSs, physical and software components are deeply intertwined leading to new intellectual challenges around the reasoning of discrete events and their effects on low-level physical dynamics.

2.2 CPS Properties

The tight interdependence of physical and computational processes give rise to fundamental properties that are shared across a diverse set of cyber-physical systems. The fundamental properties listed below are the source of many of the key intellectual challenges that define the field of cyber-physical systems.

- Reactive. A CPS interacts with the environment in an ongoing manner. A feedback loop between its inputs, which measure the environment (via sensors and outputs), and outputs, which influence the environment (via actuators), allow a CPS to react to changes. Consider the cruise control functionality in a car. The cruise control constantly monitors speed and adjusts engine throttle so that the car's speed stays close to the desired value. In this manner, the car knows when to accelerate if going up a hill.

- Physically Bounded. A CPS is bound by the physical world in which they operate. First, the time in which a CPS can react is constrained predominantly by how quickly it can effect the physical phenomenon being controlled. The reaction time of a CPS's control loop varies greatly depending on the system, but is usually on the order of milliseconds to seconds. Second, there may be physical limitations that cannot be exceeded. In the case of a car's cruise control, even if the desired speed in set to 200 mph, the car's engine may not be capable of reaching the desired speed.

- Multi-Model. The design of controllers for the physical world requires modeling the dynamics of physical quantities: to adjust the throttle, a car's cruise control needs a model of how speed changes with time as a function of the throttle. Traditionally, this would be modeled using *analog* continuous time. In a cyber-physical system, the design of controllers is complex; *digital* discrete time computing components must interact with analog models making computation more expensive. Multi-model systems such as CPS are often referred to as *hybrid systems*.

- Observable. A CPS's effect on the physical environment is observable by itself and others. For instance, a car's cruise control cannot obscure the fact that it makes a car accelerate from other sensors in the system or from an external observer. In particular, the observability of a CPS corresponds to the possibility of determining the state of the system on the basis of sensed inputs.

- Error Tolerant. The state observed by a system is a small snapshot of the true continuous analog model. The physical quantities that need to be sensed are continuous whereas digital systems are discrete. The mismatch of measurement domains requires conversion from continuous data to discrete data which introduces quantization error. Generally, systems must be resilient to errors that stem from interpolating these values to estimate the true state. Moreover, these state snapshots have to contend with environmental noise due to the nature of various sensors. As a concrete example, while quantities such as pressure and temperature can in theory have infinite precision, the precision of digital readings is limited by an analog-to-digital converter, which is typically 8-12 bits. Thus, CPSs are built with a high level of tolerance to account for uncertainties in estimation and measurement.

2.3 Framework

Cyber-physical systems come in many forms: from a car, to a robot, to the HVAC in a building. However, they share key properties and structures that can be distilled into a common framework. With a standardized framework, key stakeholders involved in the realization of a CPS can communicate and collaborate more effectively to identify and enumerate both vulnerabilities and safeguards. Previous frameworks have focused either on specific CPS classes [48, 49, 50], or on formal models for verifying threats [51, 52]. Existing software security frameworks are either too abstract or fail to capture the extra dimensionality of security threats that emerge from the composition of components that make up a CPS [53]. By not taking into account the way components interact, CPS stakeholders are handicapped in their ability to anticipate and counter threats.

The proposed framework attempts to unify various concepts presented in the existing literature [5, 54, 55, 56] into an ontology that captures the relevant flows of energy and information in a CPS. The ontological framework serves as a concrete tool for which to begin the threat modeling process as it helps capture key interaction points and flows across the distinct layers of a CPS. The model aims to be structured to provide a meaningful representation of CPSs, yet flexible enough to be extendible. Figure 2.1 summarizes the proposed

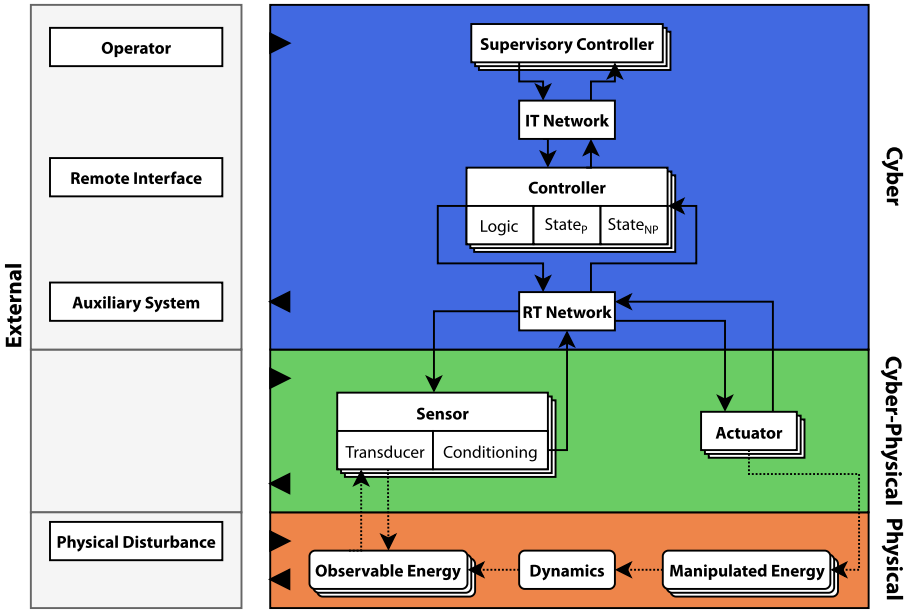


FIGURE 2.1: CPS Ontological Framework

ontological framework. In Chapter 4, we will more broadly elaborate on prominent CPS security threats and potential mitigations utilizing this framework.

WHEN IS A SYSTEM CATEGORIZED AS IOT? In the proposed framework, a CPS is considered an IoT if it does not have an actuation component and thus has no feedback loop between the cyber and physical layers. For instance, devices that only monitor temperature (e.g., a weather station) only observe changes in the environment; thus, are considered to be IoT. In contrast, a thermostat controlling an HVAC, while also monitors temperature, has the ability to physically modify the environment by actively cooling or heating an element; thus, an HVAC is considered a CPS. Popular and common IoT are devices such as Amazon's Echo, Nest's smoke alarm, and Ring's doorbell.

Under the proposed framework and categorization of IoT, it is important to note that devices that some audiences normally consider to be IoT such as Philip's Hue smart lightbulbs are in fact CPS since a controller "actuates" a light on or off. By categorizing these types of devices as CPSs, the framework can highlight threats that may exfiltrate data through the physical domain [25].

Additionally, it is important to note the distinction between systems used solely for non CPS/IoT functionality and those that exhibit CPS/IoT interactions. For example, a refrigerator with an integrated device for web browsing, that also happens to keep your food cold is not a CPS/IoT. While the refrigerator has latent

CPS/IoT components and capabilities, because there are no information/energy flows across layers it cannot be categorized as CPS/IoT.

2.3.1 Cyber Layer

The cyber layer is concerned with analyzing the information flows in the digital domain. The cyber layer is made of up of *controllers* that are connected by a traditional *IT network* such as ethernet, Wi-Fi, bluetooth, etc. A real-time *controller* is used to manage time sensitive tasks that directly influence the stability of the physical process. Logically, a controller is comprised of three main components: (i) logic — algorithms that estimate state and determine actuation commands (ii) persistent state ($State_P$) — data such as configuration parameters, cryptographic keys, etc. and (iii) nonpersistent state ($State_{NP}$) — data constructed from observing the cyber-physical layer. Most importantly, a controller serves as the bridge between the cyber and cyber-physical layers connected via a *real-time network* such as CAN, FlexRay, etc. A secondary, *supervisory controller* may commonly be present, either as a non real-time task or as a separate physical device. Usually, the supervisory controller provides supplemental functionality such as maintaining higher-level operational goals (e.g., system efficiency, navigation, etc), or a human machine interface (HMI) for an operator. Depending on the role of the human, they can be considered as part of a CPS (i.e., human-in-the-loop) or as an external entity. An in depth discussion on the software and hardware that make up the cyber layer is provided in Chapter 3.

2.3.2 Cyber-Physical Layer

The cyber-physical layer sits at the boundary between the information (cyber) and energy (physical) domains. A *sensor* translates physical phenomenon (i.e., the *observable energy*) into digital signals that can be interpreted by a computer. For example, a light sensor is a device whose resistance varies according to the intensity of light it is subjected to. Software then operates on the binary representation (e.g., a signed integer) of the intensity rather than the direct physical or electrical quantities. An *actuator* works in the reverse direction of a sensor. It takes an electrical signal and turns it into a physical action (i.e., the *manipulated energy*).

Sensors can be broadly categorized according to how energy is observed: (i) passive sensors (e.g., a thermometer) measure naturally emitted energy and (ii) active sensors (e.g., sonar) require stimulating the physical layer to trigger observable energy. These signals are received/transmitted by a *transducer* that converts

the energy from one form into another (most commonly into an electrical signal). Ultimately, the observable energy is used to extract some desired state (i.e., the measurand). Depending on the state being measured there may be multiple energy sources that can be used to extract state. For example, the relative distance to an object can be measured using reflected sound (e.g., sonar) or light (e.g., LIDAR). The signal from a transducer is fed through what is commonly referred to as the *signal conditioning chain* that is meant to reduce noise while amplifying useful information [57]. Standard components of a signal conditioning chain include amplifiers (to increase the signal gain), filters (to remove noise) and Analog-to-Digital (ADC) converters (to digitize the signal). We provide a brief overview of common CPS sensors and actuators in Chapter 3.

2.3.3 Physical Layer

The physical layer deals with the transfer of energy and how it affects the *dynamics*, or the evolution over time, of a physical process. The function, or input/output relation, of a physical process is often described by an ordinary differential equation model from physical or chemical laws, by difference equations, or Markov models. The input to the physical process is the energy manipulated by the actuator and the output is the observable energy measured by sensors.

2.3.4 External Layer

Finally, the external layer provides a mechanism to capture additional entities that influence the behavior of a CPS. For instance, a *physical disturbance*, such as snow or ice on a road surface, can affect how a car adjusts traction. A *remote interface* can be used to provide monitoring, or control functionality for an operator. Additionally, an *auxiliary system* may provide/receive signals used to make decisions. The input (▶) and output (◀) ports in Figure 2.1 are used to allow for flows between the external entities and the internals of a CPS. The bi-directionality of these ports implies that not only is a CPS vulnerable to the externalities of the environment, but the environment is equally susceptible to any hazards a CPS might introduce. Hazards introduced by a CPS can lead to the physical harm of individuals, the destruction of critical infrastructure, and the degradation of the natural environment.

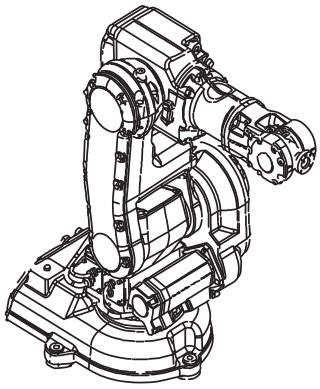


FIGURE 2.2: The ABB IRB 140 industrial robot arm.

2.4 Framework Case-Study: Industrial Robot

In this section, we demonstrate how the framework in this chapter can be applied to accommodate a concrete case study, an industrial robot arm. An industrial robot, as defined by the ISO 8373 standard, is an electro-mechanical system composed of mechanical multi-axis "arms", a control system, an "operator interface", and its hardware software communication interface. These robotic arms play key roles in various manufacturing industries with the International Federation of Robotics (IFR) projecting almost 4 million units will be employed in factories globally in the years to come [58]. We analyze the ABB IRB 140 industrial robot (shown in Figure 2.2) as described in [59]. The case-study is summarized in Figure 2.3.

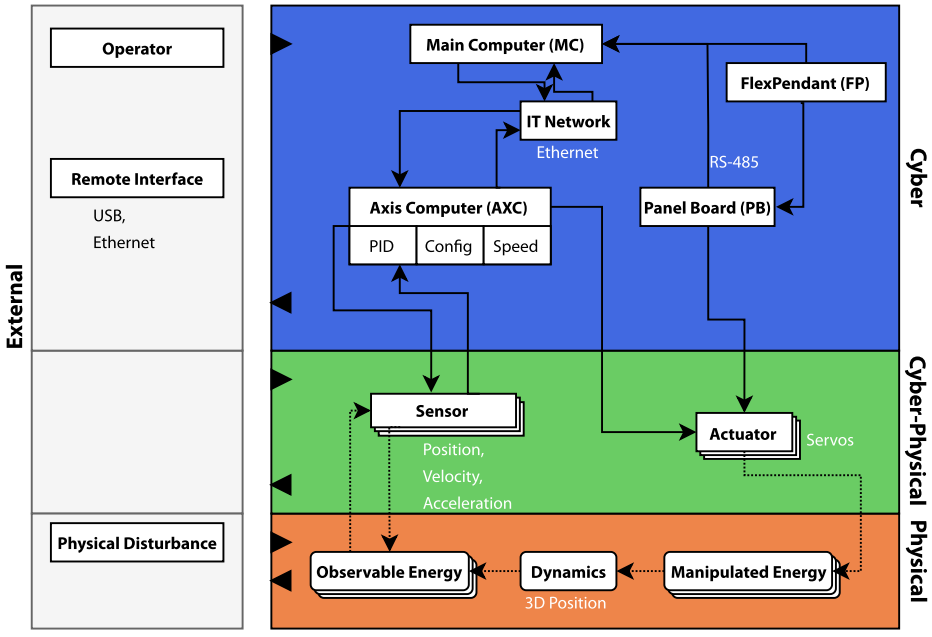


FIGURE 2.3: The industrial robot arm mapped according to the proposed framework.

Physical Layer

The dynamics of the IRB 140 industrial robot are quite straightforward. The arm has multiple joints that allow it to move in a three dimensional environment. Moreover, the arm is terminated by what are referred to as end effectors, or attachments that can be mounted to the arm for various tasks (e.g., pliers, a cutter, laser, etc.).

Cyber-Physical Layer

While the specifics of the internal sensors used by the IRB 140 are not published, we can reasonably assume that it includes a set of accelerometers, gyroscopes, and proximity sensors to detect the state of the arm. Similarly, for actuators, we can reasonably assume there are a number of servo motors that are used for proper arm articulation, in addition to the actuators that may be part of the end effector.

Cyber Layer

The control system is split into different modules: (i) Main Computer (MC) — A supervisory controller that is responsible for orchestrating the execution of tasks and coordinates the arms components optimizing for the best control and path planning strategy, (ii) Axis Computer (AXC) — The real-time controller implementing the main control loop and which interfaces with the sensors and actuators for direct control over the arm and (iii) Panel Board (PB) — A separate real-time controller, in addition to the AXC, whose main responsibility is to block the robot from violating safety-critical conditions.

Internally, the MC and AXC communicate through a standard IT network (i.e., ethernet) where they exchange any data needed by the MC for planning. The PB is connected to the MC via RS485 were it sends a periodic heartbeat packet containing the status of the robot. The AXC connects to the servos and actuators directly via a relay of electrical switches. Additionally, these systems provide a control interface, called the Flex Pendant (FP), that allows an operator to manually move the arm. The FP is connected to the PB and MC.

External Layer

The MC exposes a number of remote interfaces used for maintenance and programming purposes. The MC includes an ethernet port that can be used to connect it to a local network providing services such as FTP, USB ports to mount mass-storage devices, and various other switches to change operating modes. Physical

disturbances are always a possibility depending on where the robot arm is situated. For instance, in a multi-robot configuration one may have to consider collisions with/from other arms.

Identifying Threats

Using the proposed framework, we can identify key entities that deserve closer inspection. The main computer stands out as it is the most exposed component of the robot which serves as the central gateway for remote interfaces. As evaluated in [59], the MC suffers from a number of security issues ranging from weak cryptography to memory corruption. In addition to the MC, the framework also makes apparent the Flex Pendant and physical disturbances as possible threats. For instance, control over the FP provides an entry point to the panel board which can essentially override any control commands provided by normal MC to AXC pipeline. Similarly, physical disturbances can cause sensors to misbehave in ways the cyber layer may not be designed to cope with.

3 Characterizing Cyber-Physical Systems

This chapter takes an in-depth look at the typical sensors, actuators, hardware, and software that characterize CPSs.

3.1 Sensing & Actuation

Several classifications of sensors and actuators have been made in the past [60]. This section provides a closer look at components found in a wide array of CPSs to understand the level of physical interactivity that is typically entailed.

✦✦✦

Acceleration & Rotation

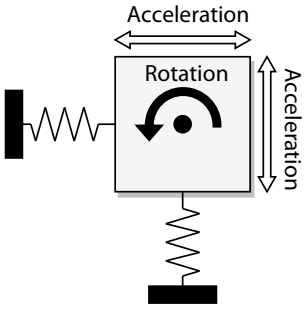


Figure 3.1: Principles behind accelerometer & gyroscope functionality.

An *accelerometer* is used when there is a need to measure acceleration (i.e., linear motion) or tilt. Accelerometers work on the principle of a mass on a spring as shown in Figure 3.1. When the object they are attached to accelerates, the mass causes the spring to stretch or compress creating a force which corresponds to the acceleration. Modern constructs of these sensors are known as MEMS accelerometers. MEMS accel-

erators are extremely common even outside of CPSs and can be found in every modern mobile device (e.g., allowing you to tilt the orientation of the display).

Closely related to accelerometers are *gyroscopes* that measure changes in orientation (rotation). They are mostly implemented together with accelerometers due to their close functionality. Gyroscopes extend the accelerometer by adding additional springs to measure rotational motion.

Positioning & Velocity

In theory, given a measurement a of acceleration over time, it is possible to determine the velocity and location of an object. Gyroscopes and accelerometers are often combined to this end where position is estimated by dead reckoning. Dead reckoning starts from a known initial position and orientation, and then uses measurements of motion to estimate subsequent position and orientation. Such units are subject to drift, so they are often combined with other methods of directly measuring position.

Global navigation satellite systems (GNSS), or more commonly known as GPS, offer a fairly accurate method of directly measuring position using trilateration [61]. A receiver listens for signals from four or more satellites that carry extremely precise clocks. The satellites transmit a signal that includes the time of transmission and the location of the satellite at the time of transmission. The receiver can thus calculate its distance from the satellite using the speed of light. Given four such measurements, the receivers position can be calculated.

Proximity & Object Detection

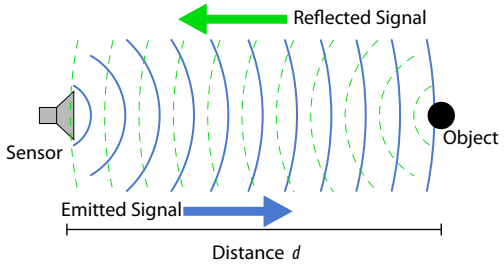


Figure 3.2: Principles behind proximity and object detection sensor functionality.

Proximity sensors are able to detect the presence of nearby objects without any physical contact. Often they provide a physical stimulus (e.g., sound or light) that reflects in the presence of an object. The changes in the reflected stimulus are then used to calculate the distance, or proximity, to a given object as shown in Figure 3.2.

These sensors are commonly found in automotive CPSs in the form of parking sensors mounted on bumpers to sense the distance to nearby objects. Proximity sensors are one of the main enablers behind the push for autonomous vehicles.

✦✦✦

Motors

The diversity of actuators (e.g., electrical, electromechanical, electromagnetic, hydraulic, or pneumatic) is extensive. The DC electrical motor is perhaps the simplest type of motor which is widely used in many CPS applications such as rotating pumps, fans, compressors, wheels, etc. Their speed of rotation can be easily controlled by varying the applied electric current. DC motors can be found on everything from robotic manufacturing arms, a car's windshield wiper, and a drone's rotors. We explore DC motors in more detail in the context of YOLO (see § 5.4) where we discuss the mathematical models used to describe them.

3.2 Control

Control theory has developed an extensive collection of mathematical models and software tools for the analysis of CPSs. By design, a CPS forms a feedback loop encompassing the sensors, the physical process, the actuators, and the controller; thus, making the concept of *feedback* central to all CPSs. Systems with feedback are referred to as *closed loop*, while those without it are referred to as *open loop* [62]. Generally, feedback allows a system to be insensitive to both external disturbances and to variations in its individual elements; something that open loop systems cannot easily accomplish. In other words, feedback provides robustness to uncertainty making it possible to design precise systems from imprecise components. The key issues in designing controllers and their corresponding algorithms is ensuring that the dynamics of the closed loop system are stable (i.e., bounded disturbances give bounded errors) and that they operate as desired (i.e., reject unwanted disturbances, respond quickly, etc).

Virtually all modern control algorithms are implemented in software. The embedded computing environment they are designed for is markedly different from traditional GP computing to address issues with timeliness and uncertainty. The diversity of processor architectures, core families with different capabilities, and operating systems perhaps makes embedded devices one of the most diverse segments of computing. In this section, we establish an overview of the capabilities of common hardware and software used in CPSs.

3.2.1 Computational Resources

The concept of a resource constrained device relates to limitations imposed on processing power, memory, and energy. However, there are no well-established thresholds for these constraints. The closest established notion of resource constraints is a classification, which we adopt, presented by the Internet Engineering Task Force (IETF) community to help the standardization of node networks [63]. The IETF proposed classifications for constrained devices considering the size of available memory, energy limitations, and strategies for power usage and network communication. For the purposes of this dissertation which revolves around software security, we focus our attention on memory. Table 3.1 shows the IETF classification. Class 0 devices have severe constraints to communicate securely and are typically pre-configured to connect to management gateways. Class 1 and Class 2 devices can usually support full protocol stacks (e.g., HTTP, TLS, etc) and tend to perform computational tasks fundamental to CPS operation (e.g., motor control, power regulation, etc).

Class	Data Size (eg. RAM)	Code Size (eg. Flash)
C0	<<10 KB	<<100 KB
C1	~10 KB	~100 KB
C2	~50 KB	~250 KB

Table 3.1: IETF Classes of Constrained Devices [63]

In what follows, we discuss how the typical computational resources (i.e., hardware) that are dominant in the CPS domain fall into the classes of constrained devices defined by the IETF. We will look at the prevalent characteristics of typical processors and memory in addition to discussing commonly available security primitives.

✦✦✦

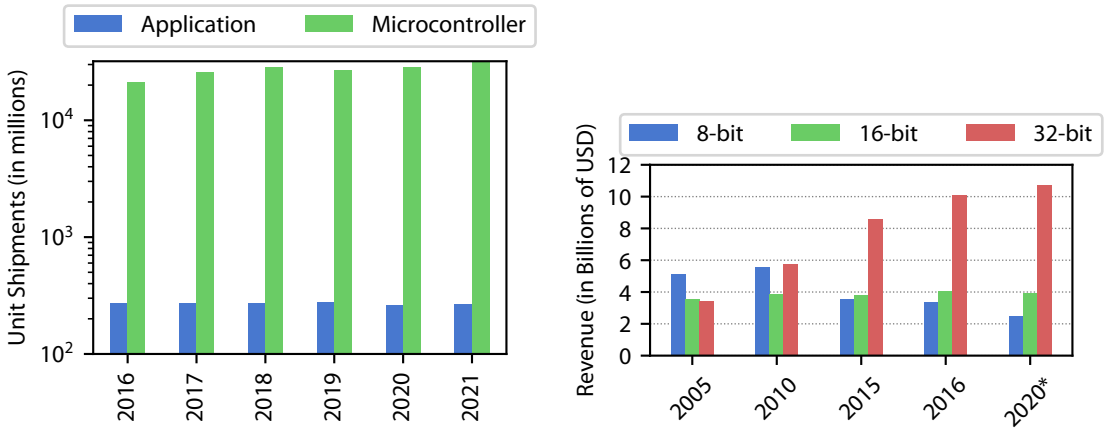
Processor Classes

The GP computing world is overwhelmingly dominated by Intel's x86 architecture. While there is no such dominance in embedded computing, ARM has been slowly engulfing marketshare with their 32-bit and 64-bit processor IP [64]. Processor families can be roughly divided into two classes according to their functionality: (i) application level processors and (ii) microcontrollers (μC).

Application Processors Application level processors come in both 32-bit and 64-bit variants and are what most people are familiar with. These processors are found in mobile phones, desktops, and servers. Application level processors are generally high performance usually operating in the >1GHz range and supporting multiple gigabytes of memory that are typically externally provided via DRAM modules. The distinguishing feature of these chips is a memory management unit (MMU) enabling virtual memory and the isolation primitives inherent to it. Application processors are capable of running GP commodity operating systems like Linux, Android, Windows, MacOS, etc. While you may find application processors in certain CPSs, they are typically used in a supervisory capacity or for offloading heavy computational workloads (e.g., vision processing).

Microcontrollers Modern microcontrollers predominately use 16-bit and 32-bit architectures [15, 16, 65, 66]. They are the dominating class of processors found in many CPS applications. In contrast to application processors, the main distinguishing characteristics are: (i) lower operating frequencies on the order of 100-500MHz for reduced power (ii) integrated memory and peripherals and (iii) the replacement of an MMU (i.e., enabling virtual memory) by a memory protection unit (MPU) for more deterministic response times resulting in a flat memory model with limited isolation primitives. Microcontrollers are typically used to provide safety-critical functionality in CPSs which require strict real-time characteristics.

Microcontroller Pervasiveness



(A) Unit Shipments between Application and Microcontroller class processors. [67, 68, 69]

(B) Microcontroller revenue by processor architecture type. [70]

Figure 3.3: Microcontroller Market Analysis [71]

Microcontrollers make up the richest processor offerings by semiconductor manufacturers targeting CPSs. For instance, NXP, a leading semiconductor company, markets their microcontroller LPC family for everything from power management, elevator controls, to smart meters [71]. In contrast to application processors, microcontrollers are the most pervasive computational resource available. Figure 3.3a shows that microcontrollers like those used by CPSs dwarf the amount of application processors that are shipped each year by over 2 orders of magnitude. As CPSs continue to grow more complex, so do their computational requirements. To accomodate this complexity, the growth of more featureful 32-bit μCs has started to dominate the market. By 2020, 32-bit μCs are projected to dominate over 16-bit variants by over seven billion dollars in revenue as shown in Figure 3.3b.

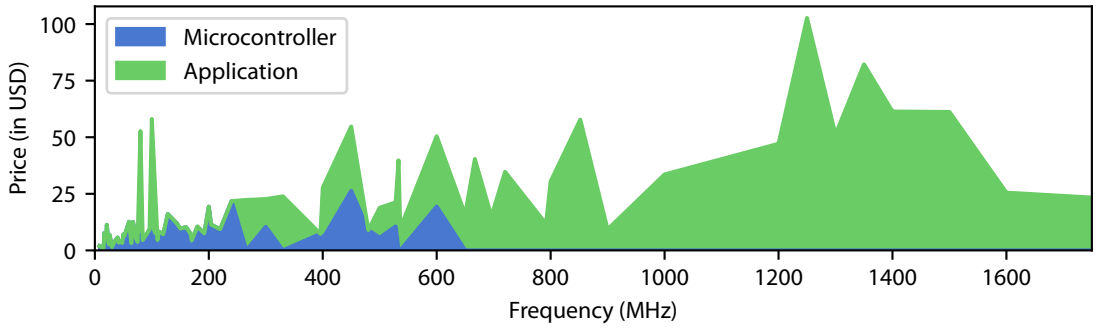


Figure 3.4: Pricing comparison between ARM 32-bit application processors and microcontrollers [72].

As previously mentioned, the cost sensitivity of a CPS is an important factor in its design. Microcontrollers not only provide determinism, but do so while providing significant savings. To put this into perspective we focus on the dominant 32-bit μC market, Figure 3.4 compares ARM 32-bit application processors (i.e., Cortex A5, A7, A8, A9, A15, and A53) and microcontrollers (i.e., Cortex M0, M3, M4, M7, M23, M33, R4, and R5) currently in production using data from Digi-Key, one of the worlds leading electronic component distributors [72]. The average price and processor speeds are plotted. As the data shows, microcontrollers are significantly less expensive than application processors.

Microcontrollers are not only cheaper than their application class bretheren, but also more energy efficient. Figure 3.5 compares different families of ARM 32-bit application processors and microcontrollers in terms of frequency and power. The results show a marked difference in power across the μCs (i.e., Cortex M0 and M3) compared to the application processors. Coupled together with their low cost, its no surprise that μCs are so pervasive.

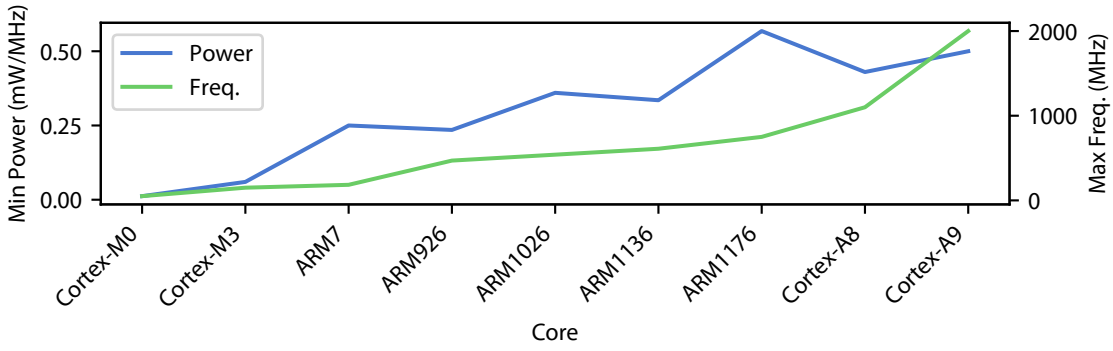


FIGURE 3.5: Power and Frequency comparison for ARM 32-bit processors and microcontrollers [73].

For the rest of this chapter, we focus on microcontrollers as they provide the majority of safety-critical functionality central to a CPS and present the most challenges for security due to their limited resources.

MEMORY MODEL

Many embedded systems are subject to varying degrees of real-time requirements. Traditionally, real-time operating systems have avoided virtual memory, and thus a MMU, due to timing analysis complications [74] opting instead for a flat memory space divided into regions.Various hardware and software designs for real-time compatible virtual memory have been proposed [74, 75, 76], but none have been readily adopted over the years. Two fundamental aspects of virtual memory have been argued to pose predictability problems regarding worst-case execution time (WCET) analysis [74, 76]:

1. *Address Translation.* The translation look-aside buffer (TLB) allows for rapid access to recent virtual to physical addresses translations. The caching of these translations inherently introduces timing variability that can make it more difficult for a developer to reason about real-time deadlines. For example, not all mappings are cached in the TLB, leading to cache misses requiring a subsequent page table lookup. Moreover, the TLB is shared among different processes leading to contention.

2. *Paging.* Predicting whether a virtual memory address results in a page fault is difficult. At any point in time, the paging algorithm may select any physical page for replacement as physical memory is shared among different processes. Moreover, page faults may incur significant overheads, rendering a system non-responsive for some period of time.

✦✦✦

Memory Hierarchy

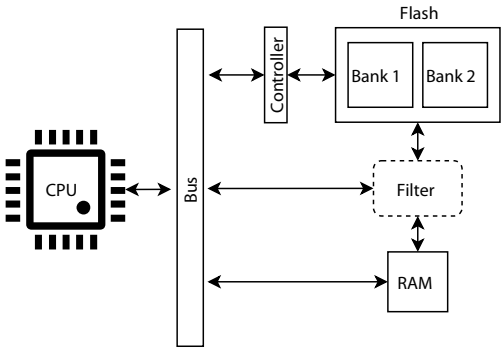


Figure 3.6: A typical microcontroller memory hierarchy.

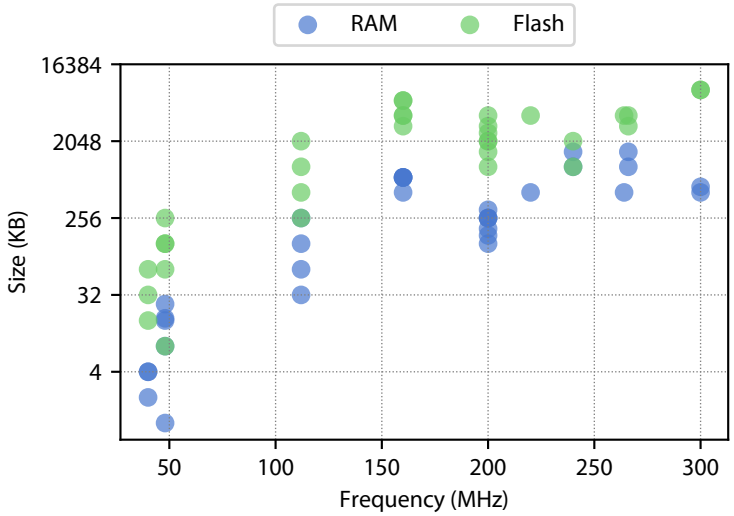


Figure 3.7: A sampling of typical microcontroller memory resources [77].

The gap between the memory of application processors and microcontrollers is drastic. The memory hierarchy usually found on systems with application processors is more complex than what is currently used by microcontrollers. A typical μC memory hierarchy is shown in Figure 3.6. Application processors usually include multiple levels of caching, RAM, and volatile memory. In contrast, microcontrollers have limited amounts of RAM and programs are usually executed directly off flash memory as opposed to being paged into RAM as in application processors. In Figure 3.7, we use NXP's line of automotive centric processors as a representative sample for typical μC memory resources. As we can see, microcontrollers usually have from

4K to 16MB of RAM while in stark contrast application processors typically have >1GB. These findings are in line with the IETF's notion of constrained devices as discussed previously. The memory limitation in μCs severely hampers what can be run on these devices necessitating efficient software.

✦✦✦

Security Features

Memory protections are of the highest importance when it comes to software security. Portions of an application might need special protection; whether it be to ensure the integrity of code and data, or confidentiality of cryptographic secrets. When containing sensitive materials, memory must not be accessible from any unexpected interface, such as a debugging port, or an unauthorized process.

Code and data on embedded systems are separated in two distinct portions of the memory hierarchy. Depending on whether code or data are to be protected, various mechanisms are available to establish protections depending on the memory type (e.g., flash, RAM, or external memory). The protection mechanisms offered by different memory are usually managed independently of each other as they are often provided by distinct vendors with limited integration. While there are number of protection primitives available, as we highlight below, it is up to the OS and the software developers to make use of them. Due to the heterogeneity of the current CPS ecosystem, software is written for the lowest common denominator and the adoption of these features varies wildly as we explore in § 3.2.2.

Memory Protection Unit The MPU is usually implemented as part of the CPU. It can prevent user applications from corrupting data used in critical tasks in the operating system kernel by changing memory access attributes. For example, the MPU can define RAM memory to be non-executable (NX) to prevent code injection. The MPU's strength lies in its ability to dynamically provide protection of a program's execution memory (i.e., stack, heap, globals). The MPU can be used to protect up to N memory regions, where N is defined by the specific architecture, preventing any malicious code from being executed from those regions. On predominant ARM families this number is 8 regions for ARMv7-M and 16 regions on newer ARMv8-M cores [78]. In addition, different privilege levels (i.e., supervisor and user modes) may be available allowing further granularity.

Memory Bus Access Filtering Certain vendors include the ability to filter memory accesses (e.g., instruction fetches, read, write operations) on the main bus as shown in Figure 3.6. The filter allows the user to establish trusted areas in the memory mapping where code or data are stored, for example between flash and RAM, and monitors access to these areas. If unexpected accesses are detected during execution an exception is generated. Accesses between these memory regions are allowed only through special designated call gates. Call gates essentially provide a very coarse-grained form of control flow integrity [79] between code and data that is split across the memory hierarchy.

Memory Tampering Protection Unlike the MPU, memory tampering features are usually implemented by flash memory. Flash memory security features are typically focused on preventing tampering or leaking of cryptographic keys and other proprietary "stored" data; they are not focused on software memory safety [80]. Flash can have externally enforceable protection mechanisms separate from the MPU/MMU requiring careful management of access control policies between the two. Vendors may implement some of the features listed below for further differentiation among competitors:

- *Readout Protection (RDP).* A global flash memory protection allowing code to be protected against copy, reverse engineering, dumping, or code injection.
- *Write Protection (WRP).* Write protection protects the contents of a specified memory area against erase or update. For instance, write protection can be enabled to prevent corruption of code or data during an update.
- *Execute-only Protection (EOP).* Memory stored in such a configured area can only be fetched by the CPU instruction bus. Any attempt to read or write this area is forbidden.
- *Hide Protection (HDP).* Memory can only be accessed only once, just after a device reset. HDP can be used to manipulate confidential data at boot. Once the application is executed, the HDP cannot be accessed.

3.2.2 Software Stack

There are a few key differences between GP and CPS software stacks. In what follows, we discuss differences in operating system (OS) abstractions and general application structure.

Operating System

Software can be written without having any underlying OS abstractions, known as bare-metal. More commonly, given the complexity of a CPS, an OS is used to handle fundamental abstractions such as scheduling, networking, and hardware. Operating systems used in the CPS domain can be classified into two groups (i) general-purpose and (ii) library OSs.

General-Purpose General-purpose operating systems are predominantly found outside of the CPS space (e.g., mobile phones, laptops, desktops, and servers). Using GP operating systems in a CPS typically involves highly tuned memory management and scheduling. Predominantly, Linux with the `PREEMPT_RT` scheduler is used for to achieve near real-time capabilities [81]. While this option is incredibly flexible and can support much of the functionality we see in the GP domain, it requires application-level processors that are more costly, and consume more power. These OSs generally require *virtual memory* capabilities as well as often being POSIX-compliant. While, there exist variants of GP kernels that target microcontrollers without a MMU (e.g., the μCLinux variant of the Linux kernel) they often have significant memory requirements (i.e., more than what is normally found in μCs) [82] and, most importantly, do not inherent the full set of features from their parents.

Library OS A library operating system provides a light weight alternative to general-purpose OSs. In a library OS, the services that a typical operating system provides are bundled in the form of libraries that are packaged together with application code to produce a single executable [83]. The majority of what are dubbed Real-Time Operating Systems (RTOS) fall under this category. A library OS gives developers the ability to select the minimum set of features required for their application to run, resulting in leaner executables that can run on resource constrained devices. Generally, these OSs do not offer *virtual memory* support resulting in a single flat address space that lacks isolation primitives found in more feature rich OSs.

As of writing, there are approximately 150 actively developed RTOSs [85]. Of the open-source ones, the most popular RTOSs ranked by GitHub contributors are shown in Figure 3.8. The data indicates that there are sizeable communities behind these projects with mbed and Zephyr dominating. We survey Zephyr as a representative OS to discuss the protection mechanisms commonly found in library OSs (and thus CPS) as discussed below.

✦✦✦

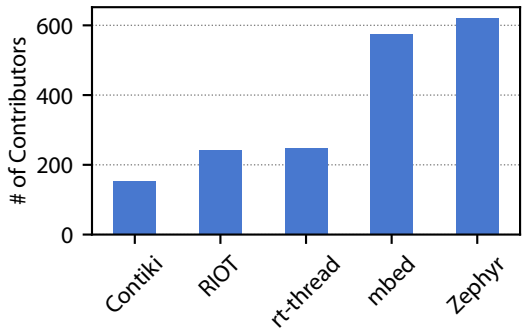


FIGURE 3.8: RTOS GitHub popularity by number of contributors [84].

PROTECTION MECHANISMS The hardware that RTOSs and library OSs are designed for impose constraints on the protection mechanisms that are provided. In distinction to GP OSs, RTOS protection mechanisms are heavily implemented during the build and boot phases. It is rare for RTOSs to implement runtime protection mechanisms due to limited resources or due to the additional overhead that may affect timeliness. Next, we discuss the common security mechanisms that help curtail memory safety issues found in library OSs and how they differ from implementations on GP systems.

- *Permissions.* A $W \oplus X$ policy on the stack is enforced with the help of the MPU. Memory can also be grouped with similar permissions. RTOSs delegate this to the build system [86] which will then coalesce memory with equivalent permissions into a single contiguous block to contend with the limited number of regions configurable by the MPU.

- *Isolation.* The isolation abstraction of an RTOS is much more constrained than on a GP OS. The userspace memory model is that of a single executable and address space. There is a notion of threads, but not full processes. User threads are considered untrusted and are isolated from the kernel and each other. Kernel threads, however, are considered trusted and have full access to each other. Memory can be shared across user threads via shared memory constructs that are configured at build time [87].

- *Overflow Mitigation.* Stack canaries, or secret values placed on the stack that are used to detect buffer overflows, are usually implemented as a compiler feature. However, due to the isolation abstraction in RTOSs additional mechanisms are provided to catch issues when the entire thread stack buffer has overflowed, not individual stack frames for which typical compiler-assisted implementations are for. Overall, the feature set provided by μCs is severely limited compared to mature application processors with primitives such as ARM PAC [88], ARM MTE [89], Intel CET [90], Intel MPX [91], etc.

- *Layout Randomization.* Randomizing the layout of code and/or data are effective means to hamper the reliability of exploits [92]. The most common randomization technique, Address Space Layout Randomization (ASLR) randomly positions the base address of an executable in the address space. Unfortunately, in the case of RTOSs this form of randomization is not possible as there is no virtual memory. Other build-time, boot-time, or execution-time techniques may be possible, but have not yet been implemented in existing OSs.

Application Architecture

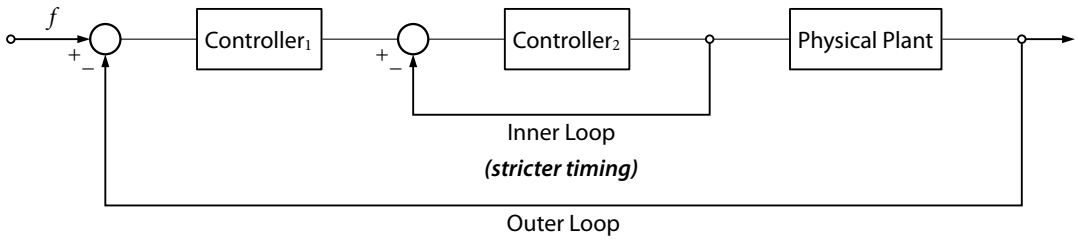


Figure 3.9: Cascade control loop structure of typical CPS applications.

Figure 3.9 illustrates the cascade control loop structure typically found in CPS control software. Cascade control involves the use of multiple controllers with the feedback loop for one controller nested inside the other [93]. Each loop has a set of actions it is responsible for performing within a given time budget. The outer loops generally have less strict timing requirements compared to the innermost loops giving the system designer more implementation flexibility. Depending on the complexity of a task and a time budget either polling or interrupt driven models may be suitable. Polling can be used to collect data from a suite of sensors on a timely basis (e.g., GPS, temperature, etc) since the resulting values may not change that frequently. Interrupts on the other hand are usually reserved for handling rapidly changing sensor data (e.g., accelerometer, gyroscope, etc), or simply to efficiently wait around on inputs you are unsure of when to expect.

4 Cyber-Physical System Security

This chapter discusses the challenges with realizing CPS security and how security differs from general-purpose systems. It also highlights various prominent threats and defensive opportunities for a CPS.

4.1 Leveraging the CPS Framework

Chapter 2 introduced an ontological framework that can be used to describe the interactions between components in CPSs. In this section, we will leverage this framework to discuss the compositionality aspects of security for CPSs.

4.1.1 Individual Component Security

In the framework presented in Figure 2.1, an individual component refers to an entity (e.g., the controller) and its respective inputs and outputs. The security of an individual component can often be analyzed independently of interactions with the rest of the system. Enhancing the security of individual components is often easier to reason about due to their limited scope or threat model. For instance, building sensors with better shielding can prevent malicious interference and adding network authentication can prevent unknown entities from eavesdropping on communication. However, while strengthening individual component security can be beneficial, within the context of the larger system it may prove to be redundant and less efficient. In other words, individual component security is analogous to finding the local minima or maxima in an optimization problem; ensuring the security of the system (i.e., what is important) requires a full system perspective.

4.1.2 Emergent System Security

Security becomes especially difficult when taking into account interactions (i.e., the composition) between multiple components within the same layer and in particular, across layers that operate in different computation domains (i.e., analog vs digital). Interactions between multiple components often leads to *emergent* behavior that requires a global perspective to reason about [94]. Reasoning about security for emergent behavior is analogous to finding a global minima or maxima in an optimization problem, requiring a full-system view. Mismatches between assumptions at the interfaces between components within and across layers can lead to vulnerabilities in the "glue" that connects them. For instance, attacks targeting controllers and other cyber components can corrupt physical state by inducing incorrect actuation commands. Similarly, attacks targeting sensors can induce corruption of cyber state (e.g., integer overflows). Properly defining the semantics of composition, which allows complex systems to be constructed from simpler components, is often non trivial. Specifying and ensuring the security of a whole system as opposed to individual components remains an open problem [95]. In response, security practitioners have resorted to a *defense-in-depth* approach, in which a series of defensive mechanisms are put in place such that if one fails, another will be in place to thwart an attack [96].

Comparison to general-purpose security

The concept of emergent behavior is not unique to CPSs. In fact, many complex cyber systems may exhibit similar phenomena between components; a CPS may leverage modeling techniques introduced for distributed cyber systems for some of its cyber components [97]. The difficulty with CPSs comes from the interactions between the discrete digital and continuous analog computational domains [98, 99]. For instance, it can be non-trivial to reason about how errors in a sensor can trigger incorrect code paths, or how data corruption at the controller can manifest physically. In other words, linking the invariants in software at the cyber layer (known to be intractable) with the invariants at the physical layer (governed by physical laws) is non-trivial. The theory of *hybrid systems* attempts to address the inherent disconnect between the two computational domains [100]. Hybrid systems theory focuses on modeling dynamics, the evolution of the system's state in time, primarily for correctness. While the hybrid nature of a CPS poses its set of modeling challenges, properly leveraged, it can prove valuable for security as we will discuss in § 4.3.

4.2 Prominent Threats

This section introduces and classifies prominent threats faced by CPSs for the respective layers introduced in the framework defined in Figure 2.1 and discusses attacks in existing literature.

4.2.1 Cyber-Physical Layer

A sensor's measurements should ideally be trustworthy as they serve to establish the ground truth about the environment. Unfortunately, sensors can be manipulated by adversaries by exploiting vulnerabilities fundamental to how they operate and observe physical phenomena. There is a diverse set of terminology used to describe the numerous acoustic, electromagnetic, and optical attacks [38, 39, 101]. In this section, we set out to further unify the nomenclature as a step towards providing a common terminology for which to discuss the threats against CPSs. Below we will describe the terminology and highlight notable attacks that have been published in the literature. A summary of notable physical threats categorized according to the proposed taxonomy is shown in Table 4.1.

Threat	Injection			Sensor	
	Type	***Energy***	***Freq.***	***Category***	***Type***
[102, 103]	Stimulus	Light	In	Passive	Camera
[21]	Stimulus	Light	In	Active	Droplet
[26]	Stimulus	Magnetic	In	Passive	Encoder
[104]	Stimulus	Acoustic	In	Passive	Microphone
[22]	Stimulus	Acoustic	Out	Passive	MEMS Gyro/Accel.
[28]	Stimulus	Acoustic	Out	Passive	MEMS Gyro/Accel.
[27]	Signal	Electromagnetic	In	Passive	Pacemaker/Defibrillator
[105]	Signal	Electromagnetic	Out	Passive	IR

Light Magnetic Acoustic Electromagnetic Passive Active

Table 4.1: A summary of threats against sensors categorized according to the proposed taxonomy.

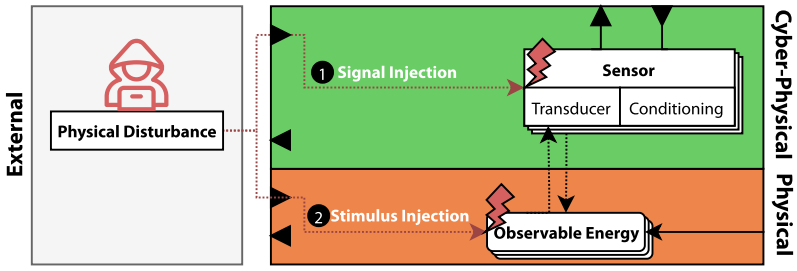


Figure 4.1: An overview of threats faced by sensors. There are two main threat vectors: ❶ signal injection and ❷ stimulus injection.

Stimulus Injection

The term *stimulus injection* refers to threats that manipulate the energy observed by a sensor (i.e., the stimulus) (see Figure 4.1 ❶). The term *injection*, as adopted by other works [27], captures the fact that the values are altered by an adversary. The stimulus signal can be modified as follows:

- In-band. An in-band injection directly alters the physical environment (e.g., acoustic, electromagnetic, optical, etc) under observation by a sensor (note that this does not involve physically tampering a sensor). Works by Davidson *et al.* [102] and Eykholt *et al.* [103] highlight how one might change the visual environment to force incorrect object detection. As these works demonstrate, this is particularly problematic for CPSs like autonomous vehicles that rely on these sensors to identify traffic signals and roadway hazards. The works of Park *et al.* [21] and Shoukry *et al.* [26] are equally problematic, but require more specialized knowledge of sensor physics to carry out. Shoukry *et al.* [26] manipulate the magnetic signal used by vehicle braking sensors that provide anti-lock braking (ABS). In a similar fashion, Park *et al.* [21] manipulate the infrared signal using an emitter to manipulate a medical infusion pump. Zhang *et al.* [104] target voice controlled systems with inaudible ultrasonic commands.

- Out-of-band. Out-of-band injections target a secondary effect that arises from how the physical phenomenon of interest is being sensed. Son *et al.* [22] exploit this attack channel by forcing resonance vibrations in MEMS accelerometers & gyroscopes. Emitting an acoustic signal at the specific resonant frequency of the sensor makes it incapable of measurement causing any CPS that relies on them to potentially malfunction. In their case, a drone that relies on these sensors is destabilized causing it to crash. Another more recent example by Bolton *et al.* [28], targets mechanical hard drives. Intentional acoustic interference causes unusual errors in the mechanics of these magnetic drives leading to damage, compromised integrity, and availability in both hardware and software.

Signal Injection

The term *signal injection* refers to threats that manipulate the resulting analog signal after it has been converted by a transducer (see Figure 4.1 ❷). In certain cases, where the sensor stimulus may be too difficult to manipulate, an attacker can resort to altering the signal anywhere along the conditioning chain before being converted to a digital value. An attacker must carefully consider where along the conditioning chain to

mount their attack in order to effectively dominate the legitimate signal. This class of attacks are also commonly referred to as intentional electromagnetic interference (IEMI). Kune *et al.* [27] and Selvaraj *et al.* [105] showcase this threat by inducing electromagnetic interference on the wires carrying the analog signal. Similar to stimulus injection, there are in-band and out-of-band injections. Signals with frequencies within the intended frequency band of the signal chain are *in-band*, otherwise they are *out-of-band*.

Comparison to other taxonomies

The taxonomy proposed by Yan *et al.* [38] limits its scope to threats that do not intentionally modify the observable energy, instead focusing on attacks that directly target a sensor's transducer (hence their terminology of *transduction attacks*). As a result, their terminology is tailored to threats arising from a sensor's design. In contrast, stimulus injection threats may require alternative mechanisms outside of the sensor's design to resolve them. An additional taxonomy by Giechaskiel and Rasmussen [39] further limits the scope of the work to only consider out-of-band injection variants. Both taxonomies can be used in combination with the one proposed here to further refine signal injection characteristics within the signal processing chain (i.e., post-transducer).

4.2.2 Cyber Layer

The cyber layer of a CPS faces many of the same threats that plague GP systems (e.g., confidentiality, integrity, availability). However, the difficulty of security in CPSs is exacerbated by the resource constraints found in typical devices. In particular, this dissertation is interested in the topic of secure software which accounts for the majority of reported CVE vulnerabilities [19] and most importantly malicious 0-day exploits [20].

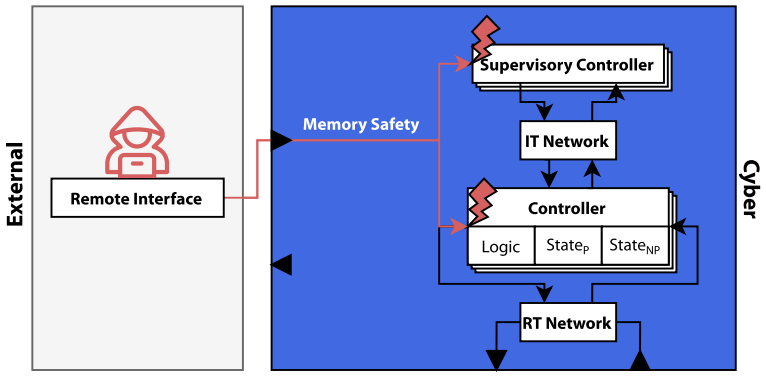


Figure 4.2: An overview of the memory safety threats. Memory safety is problematic for both supervisory and real-time controllers.

Memory Safety

Operation	No MPU	MPU
Memory Access	23	29
Context Switch	90	142

Table 4.2: Average cycle count for basic memory isolation operations with and without MPU [106].

The limitations of microcontrollers and library OSs as discussed in § 3.2.1, makes a CPS even more susceptible to memory corruption issues than a general purpose system. Memory safety vulnerabilities can target both the supervisory and main controllers of a CPS. Security primitives like the MPU have been around for more than two decades (e.g., since the ARMv4t architecture), yet it has seldom been adopted [107]. In particular, the performance overheads on memory accesses and context switches associated with the MPU make real-time constraints more difficult to meet [108]. For example, Table 4.2 shows the average cycle count for basic memory operations with and without the MPU. To put the lack of MPU adoption into context, consider that according to a recent survey [109], only approximately 33% of applications made use of memory protections. Even OSs like Zephyr and mbed that are striving to achieve parity with the protections offered by Linux are not immune memory corruption; their flat memory models impose many restrictions on isolation primitives.

The biggest security concern is that CPS controllers, especially those with limited resources, are often the weakest link in the cyber layer. Vulnerabilities in microcontrollers may serve as a prominent attack vector to exploit other parts of a system. For example, a real-time controller can be used to compromise a supervisory controller or hide information required for an operator to safely utilize the system. There have been a few documented instances of exploits that use microcontroller devices as entry points into other parts of a system: the WiFi system-on-chip on smartphones can be used to compromise the main processor [110], and adapters can be used to bypass firmware flashing mechanisms and exfiltrate data [111, 112].

4.2.3 Across Layers

The cyber-physical layer not only introduces challenges to designing a CPS, but also to securing it due to the inherent disconnect between digital and analog forms of computation. New classes of emergent threats arise from the decoupled nature of the cyber and physical layers. These emergent threats leverage vulnerabilities that span across layers to provide greater control of a system or amplify adverse effects. For instance, an at-

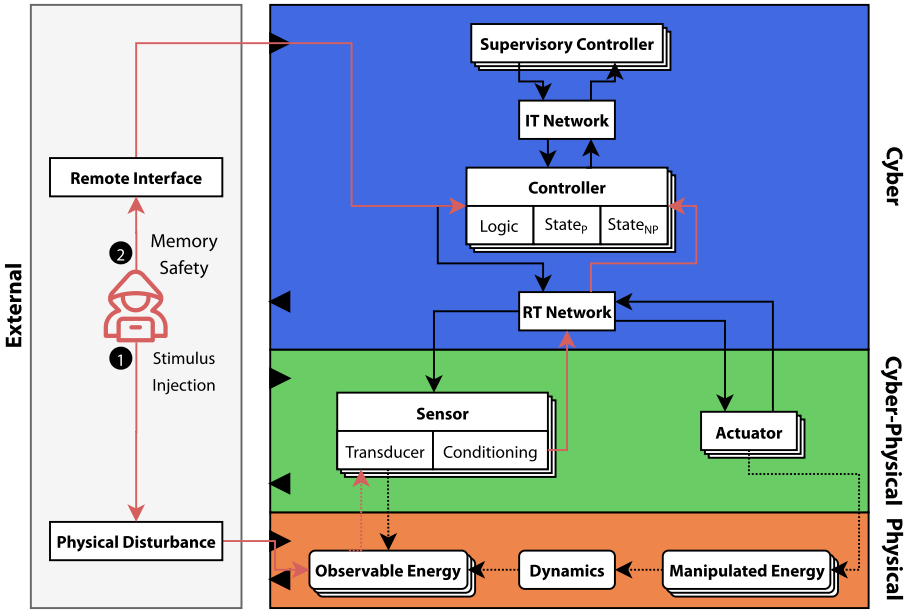


FIGURE 4.3: A cross layer threat using stimulus injection to trigger a memory corruption vulnerability. An attacker first uses a stimulus injection attack to trigger a memory safety vulnerability. Next, the attacker exploits the vulnerability directly in the controller.

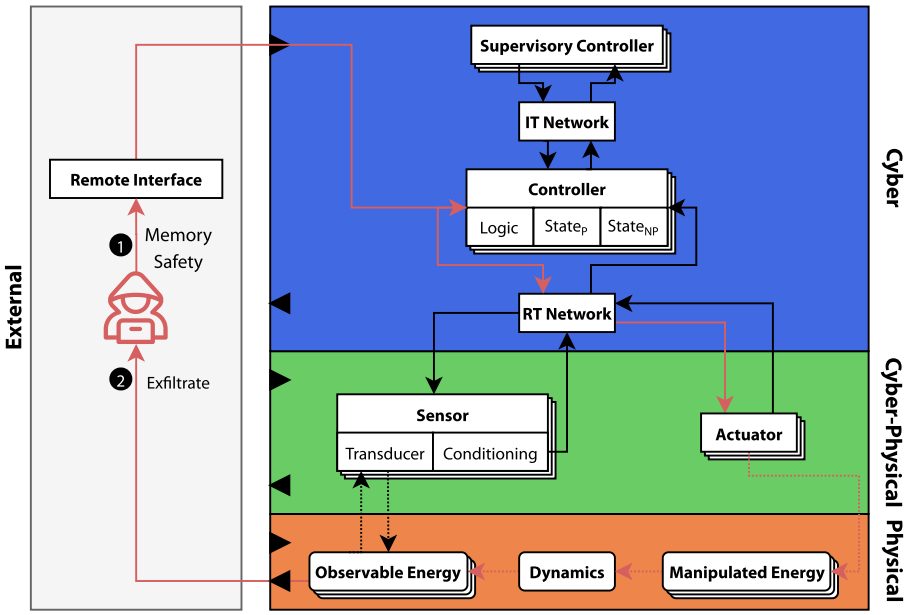


FIGURE 4.4: A cross layer threat that exfiltrates data via the physical layer. An attacker exploits a memory safety vulnerability in the controller. They then use the physical layer to exfiltrate sensitive data bypassing any IT network protections.

tacker may leverage a cyber-physical stimulus injection vulnerability to trigger a memory corruption error inside of a controller which can then be exploited using more traditional means (see Figure 4.3). Alterna-

tively, an adversary can leverage a memory corruption at the cyber layer to exfiltrate sensitive data via an actuator at the cyber-physical layer providing an effective means to bypass network isolation primitives [24, 25] (see Figure 4.4).

4.3 Defensive Opportunities

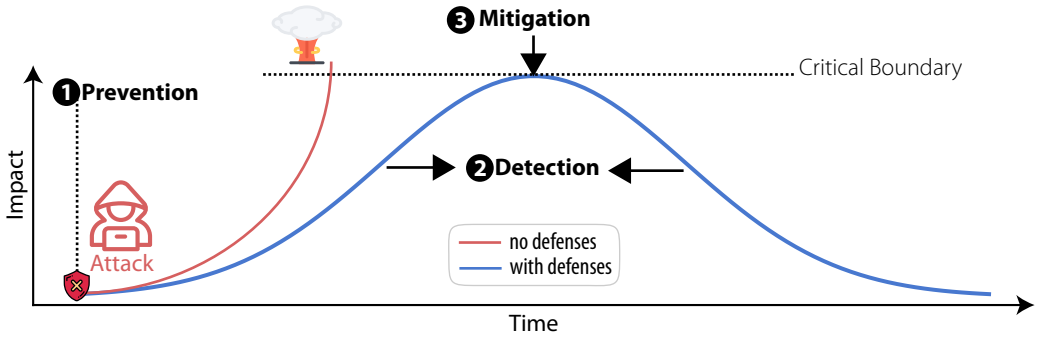


Figure 4.5: Defensive opportunities and their impact on a system over time.

CPS characteristics such as observability can make security difficult. However, observability (along with others discussed in § 2.2) can be equally useful when implementing defensive methods. As the name implies, prevention techniques guard against having an attack occur in the first place. In other words, they stop an attacker in their tracks before it has an impact on the system as shown in Figure 4.5 ❶. By detecting an adversary early, detection techniques allow a system to recover quickly limiting the impact of an attack (i.e., making Figure 4.5 ❷ more narrow). Finally, mitigations attempt to bound the maximum impact of an attack (see Figure 4.5 ❸) as opposed to letting it exceed a critical point (i.e., the red curve in Figure 4.5) that can ultimately lead to irreparable failure.

In this section, we will show how leveraging physical properties has been instrumental across various aspects of CPS security. Additionally, we will highlight how CPS constraints have led security researchers to revisit key computing abstractions to efficiently implement defenses.

4.3.1 Leveraging Physical Properties

Perhaps the most important characteristic of a CPS is the notion of physical invariants. Physics is expressible as invariant laws, for instance, Newton's laws of motion or Maxwell's equations for explaining electromagnetism. These physical invariants can be used to help define what secure, safe, and correct behavior looks

like; physical laws bound the state-space of a system essentially reducing the complexity of enforcing secure operation. In contrast, non-CPSs generally have no similar fundamental invariants to aid in defining what secure behavior looks like [113].

CPSs' physical properties have not only influenced the design of YOLO which we will discuss in Chapter 5, but a number of other defenses in existing literature [114]. Physics-based techniques have been used for the prevention, detection, and mitigation of various classes of threats; each provides a meaningful contribution to limit an attacker's impact as shown by the dashed line in Figure 4.5. We discuss techniques in each of these categories below.

Prevention For instance, to prevent false data from being injected from unknown sources, a system might employ authentication. PyCRA [115] provides a method of ensuring the trustworthiness of active sensors by comparing environmental responses to a series of physical queries or challenges. PyCRA relies on the fact that physical hardware has inherent physical delays which equally affect the system and the attacker. By randomly modulating a sensor's signal, PyCRA is able to prevent an adversary from injecting a malicious signal maintaining the integrity of sensor data.

Detection Changes in a CPS's state must follow the immutable laws of physics. For instance, the physical properties of a car (mechanical), a water plant (fluidic), or a power grid (electromagnetic) can be used to create prediction models to assert correct execution; control commands and sensor data can be checked for consistency with the expected behavior of the system naturally lending itself to an anomaly detection scheme. The idea of using physics-based models to detect attacks has been explored in existing literature; for this reason, unlike in cyber-only systems, behavior-based anomaly detection techniques are prevalent in the CPS domain [116]. Anomaly detection techniques have been applied to a variety of CPSs: water plants [117], power grids [118, 119], chemical plants [120], and other industrial control systems [121].

Mitigation Resilient control defenses aim to withstand (or mitigate) malicious attacks by taking advantage of control theoretic approaches. One of the main areas of research by the control theory community on this front involves determining which sensors are sending false information [122, 123, 124]. The main idea involves using an ensemble of sensors correlating information between them to identify subsets that are not under attack. These schemes become increasingly powerful when sensors are appropriately picked such that not all of them suffer from the same threats as discussed in § 4.2.

4.3.2 Revisiting Computing Abstractions

Security researchers have found ways to do more with less in the CPS domain. The resource constraints of CPS devices has forced researchers to revisit computing abstractions to more efficiently design defensive techniques, in particular, to protect against memory safety threats. A large body of work has focused on ways to revisit the memory layout (e.g., code and data sections) to make use of the MPU's limited protection regions to build memory safety mitigations [106, 108, 125, 126, 127, 128]. Others have found ways of porting general purpose techniques, such as a shadow stack, efficiently by leveraging microcontroller class processor features [129]. While techniques that have targeted embedded devices have focused on software implementations, as we present in Part III, PAS and Califorms revisit computing abstractions at the software-hardware interface.

4.4 Challenges

Here, we discuss some of the major challenges faced by cyber-physical systems when it comes to implementing security defenses.

4.4.1 Physical Limitations

By design, sensors are made to be sensitive to the stimulus they are measuring. Thus, an attacker can exploit this fact by directly manipulating the measured value, calling into question the trustworthiness of measurements. To address these threats it is likely that higher level architectural mitigations (e.g., redundant sensors and sensor fusion algorithms) are needed, ultimately increasing cost and complexity. In addition, imperfections in sensing components can lead to sensitivity to unintended stimulus causing undesired effects.

Realistically, there are only a limited number of ways of measuring physical phenomena. It may be too naive to think that we may be able to eliminate the security shortcomings of sensors. While building sensors with less susceptible stimulus and additional filtering are a must, there is a limit to what can be practically done without impeding a sensor's accuracy and usability. Unlike software vulnerabilities, there is usually no quick way to remedy a sensor's flaws.

4.4.2 Resource Limitations

The computational resources of many CPSs artificially limits the budget that can be allocated to security. Coupled with the nature of real-time CPS workloads, the integration of security into a CPS is particularly challenging. As discussed in § 3.2.1, the limited resources (e.g., a few MBs of Flash and hundreds of KBs of RAM) are insufficient to implement more robust mechanisms as seen in GP systems. For instance, randomizing the location of code with address space layout randomization (ASLR) is already severely limited on 32-bit architectures [130]. In general, either new security techniques that scale to resource constrained devices or the porting of existing techniques with clever ways to circumvent limitations are required.

4.4.3 Adoption

Many defense mechanisms have been proposed, but only a handful have seen widespread adoption. The reason for this can be primarily attributed to (i) high overheads/cost and (ii) compatibility with existing code.

- Cost. It is no secret that incorporating security features impacts performance. Previous work [131] surveyed mitigations against memory corruption for GP systems and found that mitigations with more than 10% overhead do not tend to see widespread adoption in production environments. The study also suggests that overheads below 5% are desired by industry. While no similar analysis currently exists for CPSs, it stands to reason that because of the additional real-time constraints these numbers may be lower.

  To give some context to the importance of performance for CPSs, previous work [108] evaluated the overhead of the MPU primitive upon which defenses usually rely on. The results show that the overheads can cause real-time constraints to be violated in some scenarios. For instance, due to the highly interactive nature of CPS applications, mechanisms that rely on the MPU can be costly due to the constant context switching that would occur from peripheral I/O operations.

- Compatibility. Compatibility with existing code also plays a huge rule in adoption as rewrites can be intrusive and costly [132]. Unfortunately, techniques that involves human intervention may be considered impractical by many. As a consequence, generally those that involve changes to limited aspects of the runtime, compilation process, or hardware are viewed more favorably.

4.4.4 Testability

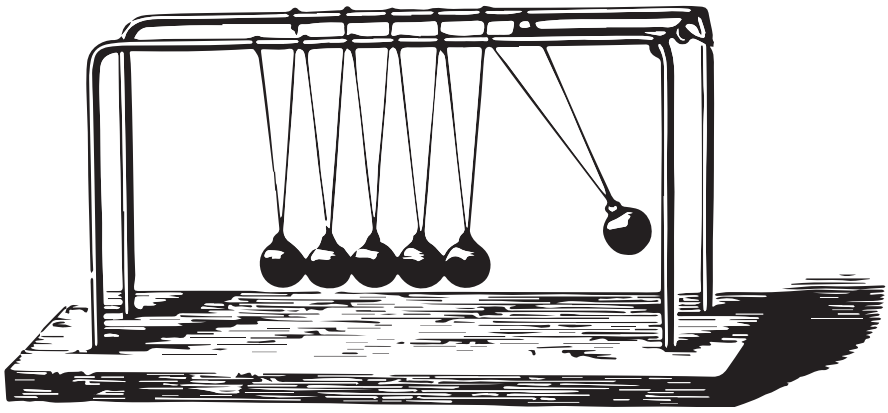
Testing is crucial to catching and eliminating software bugs. Two primary factors make testing CPSs particularly challenging in contrast to GP systems: (i) a lack of standardization and (ii) the coupling with the physical environment.

- Lack of Standardization. Typified by the lack of standardized hardware and diverse functionality, as discussed in Chapter 3, CPSs present unique challenges to system architects due to the tight coupling of software and hardware. Developers often need to tailor existing security techniques such as program analysis, fuzzing, symbolic execution, and cryptography to work on more restrictive environments. In conjunction with the lack of transparency between OEM suppliers, the obscurity of component inner workings makes testing for security less automatic than in the general-purpose domain [133, 134].

- Physical Coupling. Software systems can be tested to extreme limits without worrying about safety. Unfortunately, the same cannot be generally said about CPSs due to their physical nature. In order to take advantage of modern program analysis techniques, such as fuzzing or symbolic execution with any kind of scale or depth, system architects must have the ability to execute CPS software without inducing catastrophic failures. To fully test limits without potentially damaging expensive equipment or causing physical harm to others, system designers often rely on simulators. However, simulators may lack enough fidelity to capture complex behavior and depending on the system are rarely easily available. Moreover, simulators are rarely designed to model malicious behavior which limits their use for security testing.

Part II

Leveraging Physical Properties for Software Security



5 You Only Live Once (YOLO)

Cyber-Physical Systems provide a unique opportunity to leverage the physical environment to strengthen the security of its cyber components. As discussed in § 4.3, physical invariants can help define what correct and secure behavior looks like. In this chapter, we discuss how we can leverage physical properties to reduce the computational load required for achieving software security. By leveraging CPS properties, as discussed in § 2.2, we can develop solutions that are more easily retrofitted into legacy systems, requiring minimal modifications to software or hardware.

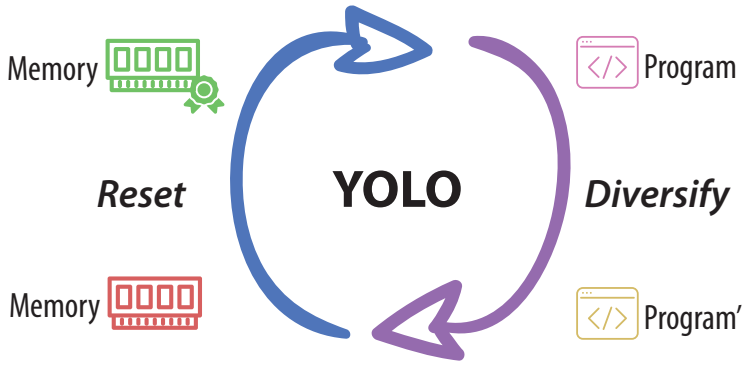


Figure 5.1: The high level YOLO approach. Memory is wiped clean on every reset and program code is diversified continously.

5.1 Overview

YOLO is a new security resiliency mechanism specifically for CPS that leverages their unique cyber and physical properties. YOLO takes advantage of *inertia*, i.e., the ability of a CPS to stay at equilibrium, and to tolerate transient imperfections in the physical world to survive attacks. You Only Live Once (YOLO) is best understood with an example. Consider an unmanned drone: even if engine power is cut off, it will continue to fly; similarly, even if a few sensor inputs are incorrect, the drone will continue to operate as the

system is designed to handle intermittent sensor errors that can occur during normal operation. In YOLO, we take advantage of these features. We intentionally reset the system periodically to clear software state that may have been manipulated by an attacker. During resets, we rely on inertia to continue operating and on diversification [135] to prevent the same attacks from breaking the system.

YOLO offers three major security benefits. First, the effects of resets, i.e, wiping memory and bringing software to its initial state, make it difficult for the attacker to persist on a system. Second, the periodicity of resets enforces a limit on the time an adversary has to carry out an attack. Finally, diversification is used to force an adversary to develop a new attack strategy across different attempts.

Realizing YOLO is not without challenges. We validate our methodology by implementing a simulation of a YOLO-ized DC Motor. Our results show that YOLO has a negligible performance effect while limiting the adversary's ability to attack the CPS. Furthermore, we experimentally evaluate YOLO on two popular stateful CPSs with varying levels of inertia, safety, and control complexity requirements . Using an engine controller unit (ECU) and measurements from a real combustion engine, we discuss and measure the performance and safety impacts of YOLO. Additionally, we also perform measurements on a quadcopter's flight controller (FC) that involves more sophisticated control. We find that YOLO-ized versions of these systems tolerate multiple frequent resets safely.

✦✦✦

Why Reset? YOLO takes advantage of a simple and universally applicable panacea for a multitude of software issues, resetting. Even among expert users, a reset is one of the preferred solutions for numerous problems in the computing world. The simple intuition behind the effectiveness of this approach is that software is tested most often in its pristine state, as discussed by Oppenheimer *et al.* [136]. With respect to the overall health of the system, the conditions of a reset provide a predictable and well defined behavior.

From the viewpoint of thwarting an attacker, the restoration of state, whether it be code or data typically helps prevent an adversary's ability to corrupt the system. The restoration of state provides increased resiliency against a large class of attacks. For example, a simple reset can remove the effects of exploits in memory. By frequently performing resets, YOLO limits the effects an adversary might have, as well as, the time needed to complete an attack. In other words, an adversary has a bounded time horizon over which they can affect the system simply because malicious effects are frequently removed.

Why Diversify? Typically, once a vulnerability is identified, an adversary can continuously carry out an attack as long as the vulnerability remains present and if they are able to keep up with the system. To remedy this, some variability must be introduced into the system; otherwise, the same vulnerability would persist. Diversification introduces randomness to prevent the system from being compromised by the same method continuously. The benefits of such an approach are successfully shown in a number of related works [135]. As a consequence of diversification, YOLO is able to lower the adversary's chance of repeated success.

Why Reset & Diversify? Diversification can help protect data that needs to be carried across resets. Some CPSs may require certain data that cannot be re-learned during normal operation (e.g., $State_P$ in Figure 2.1). For example, sensor calibration data of a quadcopter can only be obtained while it is not in flight. Therefore, state must be preserved across resets. One mechanism to protect such persistent data is to take advantage of diversification. Diversification can be used to change the location of the data on every reset, making it harder for an adversary to locate.

The benefits of diversification alone can in fact be less significant than one would expect [137]. Because these techniques rely on probability, there are cases in which they fail. Resets are used to add resiliency in situations where attacks can succeed. By proactively resetting, we can also artificially increase the diversification entropy.

Why does YOLO work for CPS? CPSs have a number of properties (as discussed in § 2.2) that allow for interesting security techniques to be explored. Mainly, YOLO leverages the fact that CPS are:

- Physical Bounded. CPSs have *inertia* — the resistance of an object to any change in its state. Inertia means that CPSs take time to react to external stimuli. This characteristic is essential for YOLO as it asserts that CPSs can continue operation and exhibit a tolerance to missed events, either by design or due to faults.

- Observable. The fact that CPSs can observe their environment using sensors is fundamental to their design and operation. Observability implies that the state of the system that caused an actuation can be relearned by the sensors. Roughly speaking, this ability to recover state via observation is the same as storing the state of the system (before actuation) to a data store (e.g., database) outside the system (the environment), and then reading it back to the system (through the sensors). Observability is critical

to YOLO as it allows for certain state to be discarded during resets, i.e., transmitted out of the system before a reset, and re-observed once out of reset.

Some of the advantages of these properties can be emulated in GP systems, but at a cost. For example, GP systems may rely on replication to emulate some of the benefits of inertia, usually with the cost of additional memory or hardware. To account for the lack of observability, traditional systems might require a secure data store or some external entity from which to recover their state. Restoring this state and ensuring consistency may be more costly than what is typically involved in the CPS domain.

What parameters can be tuned? One key challenge in applying YOLO is to determine how to safely apply resets while guaranteeing stability. We define a theoretical model by which designers can evaluate YOLO on their CPS in § 5.4. System designers can select appropriate reset & diversification strategies according to a system's constraints. At a minimum, a reset implementation should clear the running (volatile) memory of the program. Systems with more flexible reset timings should be able to tolerate more complete reset mechanisms, such as restoring code in non-volatile memory. Additionally, depending on the timing constraints of a system, different diversification techniques [135] may be applied.

5.2 Threat Model

For this work, we consider an adversary that seeks to exploit the software running on the controller via a remote interface (as described in § 4.2.2). The assumptions we make are:

- *An attacker has complete knowledge of the system internals.* The physics of the system and the control algorithms used are known to the attacker.
- *The attacker's goal is to sacrifice the integrity of the physical subsystem and bring it under their control.* The adversary's intention is to hijack the targeted system and/or prevent it from achieving its intended mission.
- *An attacker's proximity to the target is ephemeral.* An adversary may not always be within communication range; they may be limited by their equipment or other environmental factors.

We do not consider Denial-of-Service (DoS) attacks to be within scope as it goes against our assumption of having the adversary bring the system under their control. While we do not consider attacks on sensors and actuators as our defense targets software, an adversary may use them as entry points to exploit software.

5.3 Security Discussion

Unlike the traditional GP computing counterparts, cyber-attacks on CPSs encompass not only the effects of software exploits, but additionally the physical reaction to an exploit. Determining the effectiveness of YOLO involves: (a) analyzing timeliness with respect to how long an attack takes and (b) the effects of a reset on an exploit depending on where in memory the exploit resides.

We focus on two main classes of vulnerabilities an adversary is likely to exploit: integer overflow/underflow, and memory safety. Empirical studies have determined these two vulnerability classes to be responsible for nearly 80% of the recent exploits [138].

5.3.1 Memory

Depending on the adversary's goals, exploits move between different types of memory. In addition to timeliness, the conditions provided by a reset affect exploits differently depending on where in memory they reside.

Volatile

Most exploits typically begin in volatile memory. YOLO is extremely effective against these exploits. There are reasons for an adversary to not move to non-volatile memory, especially if they wish to leave as minimal a trace as possible. An ideal reset restores the entirety of memory, however, this may not be possible for all applications, especially those that need to persist state in volatile memory across reset boundaries. Depending on the complexity of the application the amount of persisted state may also vary in size. The smaller this state, the more security benefits YOLO provides. We note that persisted state can be protected with various encryption schemes depending on the size. In other words, small amounts of persisted state need not pose a significant threat.

Non-volatile

When the goals of an adversary are longer term, they typically must achieve persistence in some form of non-volatile memory. Threats such as *rootkits, spyware,* and *ransomware* all typically aim to make the transition to non-volatile memory. As previously mentioned, an ideal reset should restore the entirety of memory including the non-volatile portion. In some cases, limitations of the platform may make this impractical. For example, the limited lifetime of flash memory imposes restrictions on how often the system can be reprogrammed.

5.3.2 Timeliness

The timeliness of an attack is based on the time an exploit makes an observation of the system and the time at which an attack is completed. We use the framework developed by Evans et al. [137] to study the effectiveness of YOLO with respect to the time to attack. There are two cases to consider: (i) the time for the *exploit* and (ii) the time for the *attack*. Typically, these terms are used to refer to the same thing, but due to the nature of CPS we make an explicit distinction. The time for an *exploit* corresponds to the time needed by an adversary to hijack the control flow of software. The time for an *attack* is the time for an exploit in addition to the time for the physical system to react to an adversary's commands—their objective. *Therefore, if the reset period for YOLO is less than the time needed for the exploit, the adversary is prevented from ever exploiting the system.* Similarly, if the reset period is less than the time for an attack, the adversary is prevented from achieving their goal.

Exploit Time

We first examine the time required to exploit software. Depending on the level of control a vulnerability affords the adversary, the time required for them to mount an exploit can vary. In the worst case, they may have access to a zero-time operation i.e, an integer vulnerability used to make a security decision which can trigger the malicious behavior instantly. In this situation, YOLO does not prevent the exploit, but will rely on the side-effects of resets to provide resiliency and limit an attacker's influence.

To leverage memory safety vulnerabilities on YOLO-ized systems that use diversification, an adversary must do work proportional to their knowledge of the program layout and the strength of diversification. The strength of diversification affects how difficult it is to trigger the vulnerability and how hard it is to hijack

control flow. Assuming defenses that prevent code injection are in place, adversarial strategies can be broadly categorized into: layout blind and layout aware.

Layout blind exploits use information such as crashes to aid in searching the address space of a program for useful gadgets to craft an exploit. Given that timeliness is important to YOLO's security, let us consider the derandomization exploit by Shacham *et al.* [130] as an example to get a sense of the timescales at which resets should occur at. The system being exploited in their evaluation was a 32-bit 1.8GHz Athlon machine. Their fastest recorded time was 29s with an average of around 3.6 minutes. Consider the same scenario on a typical microcontroller running at 200MHz. As a rough estimate, if we simply scale these values by the processor frequencies, this results in a 9x slowdown turning a 29s exploit into one that takes over four minutes. As long as the YOLO-ized system can reset promptly these types of exploits can be prevented.

Layout aware exploits like Just-In-Time Code Reuse (JIT-ROP) [139] leverage pre-existing knowledge, such as the location of a code pointer, to disclose and harvest the run-time memory of the program to search for useful gadgets to craft the rest of the exploit. These exploits are difficult to practically compare given differences in code size between CPS applications and traditional applications. As proposed, published exploits leveraging memory disclosures similar to JIT-ROP target systems with a JIT engine, such as web browsers. This leaves an open question as to whether these exploits are threats to CPSs where JIT like execution is avoided due to timing and resource constraints.

Attack Time

Even if an attacker successfully exploits the system, YOLO relies on the assurances of the physical properties to resist the negative impact of an adversary until the next reset period. Given the types of attacks seen in practice, the adversary's goals usually involve compromising the physical plant. Inertia and other physical properties limit the rate of change of the system. Essentially, this means that attacks take longer as they not only involve the exploit, but the time an adversary requires to influence the system to meet their goals. For example, Urbina *et al.* [140] discuss metrics and analyze how these physical properties are used by intrusion detection techniques to place bounds on the rate an adversary disturbs the system. The authors found that if the adversary's rate of attack can be made sufficiently low, the physical properties will allow the system to resist the negative effects of the attack. In turn, this means the system still meets its original objective despite the presence of an adversary.

5.4 Theoretical Analysis

In this section, we introduce an analytical approach to evaluate the feasibility of a YOLO-ized system. We define the constraints on reset periods in order to maintain the CPS's stability.

5.4.1 Problem Formulation

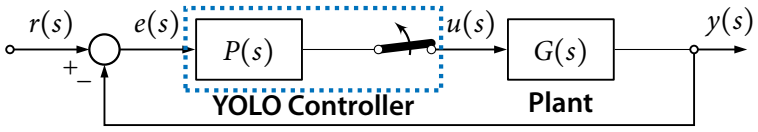


Figure 5.2: The closed loop model of a CPS. A physical plant with a transfer function $G(s)$, and the YOLO-ized controller, with a transfer function $P(s)$.

The framework shown in Figure 5.2 consists of a closed-loop system, which includes a physical plant and a YOLO-ized controller. The plant has a transfer function $G(s)$, while the transfer function of the controller combined with YOLO is represented by $P(s)$. YOLO acts as an ON/OFF switch that resets the controller periodically. To formulate this switching behavior, we refer to the time interval between any two successive resets as T_R. T_R is composed of two main components as shown in Eq. (5.1).

$$T_R = T_u + T_d \tag{5.1}$$

where T_u is the controller up-time, in which YOLO is inactive, and T_d is the controller down-time, in which the controller is out of service due to the YOLO's effect.

Based on YOLO's state, the closed loop transfer function of Figure 5.2 takes two different forms. The first form occurs when YOLO is inactive (i.e. the controller output is directly connected to the plant input, during T_u), as shown in Eq. (5.2). The second form occurs when YOLO is active (i.e, the controller is in the reset mode, during T_d), as shown in Eq. (5.3)

$$F_{inactive}(s) = \frac{y(s)}{r(s)} = \frac{P(s)G(s)}{1 + P(s)G(s)} \tag{5.2}$$

$$F_{active}(s) = \frac{y(s)}{u(s)} = G(s) \tag{5.3}$$

where $y(s)$ and $r(s)$ represent the closed loop system output and the input reference, respectively. $u(s)$ is the plant input signal. This system is viewed as a switched system [141], a hybrid dynamical system composed of a family of continuous-time subsystems with rules orchestrating the switching between the subsystems. The state space model of our switched system is given by Eq. (5.4).

$$\begin{cases} \dot{x} = A_i x + B_i r, \\ y = C_i x + D_i r \end{cases} \tag{5.4}$$

where $x \in \mathbb{R}^n$ is the state, $r \in \mathbb{R}^m$ is the reference input, and $y \in \mathbb{R}^p$ is the system output. The matrices A_i, B_i, C_i, and D_i are the system matrices, where $i \in \{1, 2\}$. By representing Eq. (5.2) and Eq. (5.3) using the general n^{th} order transfer function Eq. (5.5), we define the main matrices using the controllable canonical form.

$$F(s) = \frac{b_0 S^n + b_1 S^{n-1} + \cdots + b_{n-1} S + b_n}{S^n + a_1 S^{n-1} + \cdots + a_{n-1} S + a_n} \tag{5.5}$$

$$A_i = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -a_n & -a_{n-1} & -a_{n-2} & \cdots & -a_1 \end{bmatrix} \qquad B_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$C_i = \begin{bmatrix} (b_n - a_n b_0) & (b_{n-1} - a_{n-1} b_0) & \cdots & (b_1 - a_1 b_0) \end{bmatrix} \qquad D_i = b_0$$

For consistency, we refer to the case, in which YOLO is inactive ($F_{inactive}$) and active (F_{active}), by the index $i = 1$ and $i = 2$, respectively.

5.4.2 Stability Analysis

To prove that the system is safe under YOLO's resets, the stability of the switched system shown in Eq. (5.4) should be guaranteed. There exists a considerable amount of work in control theory literature, which investigates the stability of switched systems [142]. These works mainly depend on Lyapunov functions, a method

used to prove the stability of dynamic systems without requiring the actual solution of the system's ordinary differential equations to be available [141].

Among the available stability analyses, we adopt the average "dwell time" approach proposed by Zhai *et al.* [143] that maintains the stability of a system containing Hurwitz stable (i.e., all eigenvalues lie in the left half complex plane) and unstable subsystems. This configuration maps to the *inactive* and *active* forms of YOLO. The "dwell time" concept is defined as the time between consecutive switchings [142] and represents the minimum boundary that should be respected to guarantee stability. The average "dwell time" allows some consecutive switching to violate the dwell time constraint as long as the total average switching time is no less than a specified constant, τ [142].

Zhai *et al.* [143] show that if τ is chosen sufficiently large and the total activation time of unstable subsystems (T_d) is relatively small compared with that of Hurwitz stable subsystems (T_u), then exponential stability of a desired degree is guaranteed. The stability conditions are shown in Eq. (5.6) and Eq. (5.7), respectively (see [143] for further details)

$$\tau \geq \frac{a}{\lambda^* - \lambda} \tag{5.6}$$

$$\frac{T_u}{T_d} \geq \frac{\lambda^+ + \lambda^*}{\lambda^- - \lambda^*} \tag{5.7}$$

where λ^- and λ^+ are positive scalars representing the minimum and maximum eigenvalues of the stable and unstable subsystems, respectively. For any given $\lambda \in (0, \lambda^-)$, λ^* is chosen arbitrary, where $\lambda^* \in (\lambda, \lambda^-)$. Finally, a is the maximum of a set of scalars a_i, which satisfy the following condition [143].

$$\begin{cases} \|e^{A_i t}\| \leq e^{a_i - \lambda_i t} & i \in \mathbb{S} \\ \|e^{A_i t}\| \leq e^{a_i + \lambda_i t} & i \in \mathbb{U} \end{cases} \tag{5.8}$$

where $\mathbb{S}$ represents the stable subsystems and $\mathbb{U}$ represents the unstable ones. In our scenario, switching times, T_u and T_d, are fixed. Thus, to maintain stability, $(T_u + T_d)/2$ should be greater than τ. By using the plant and controller parameters, given in Eq. (5.2) and Eq. (5.3), to solve for τ and T_u/T_d, given in Eq. (5.6) and Eq. (5.7), with the help of the above boundary rule Eq. (5.8), a piecewise Lyapunov function ($V(x) = x^T P_i x$) [141] is constructed to ensure the global exponential stability of the system [143]. P_i are positive

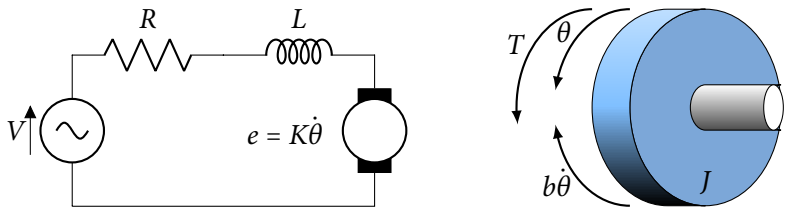


FIGURE 5.3: Simple abstraction of a DC motor. The voltage source, v; the rotational speed of the shaft, $\dot{\theta}$; the moment of inertia of the rotor, J; the motor viscous friction constant, b; the electromotive force & motor torque constants K; the electric resistance, R; and the electric inductance, L.

definite symmetric matrices, $P_i \in \mathbb{R}^n$, which are directly obtainable by solving the linear matrix inequalities (LMIs), given by Eq. (5.9).

$$\begin{cases} (A_i + \lambda_i I)^T P_i + P_i (A_i + \lambda_i I) < 0 & i \in \mathbb{S} \\ (A_i - \lambda_i I)^T P_i + P_i (A_i - \lambda_i I) < 0 & i \in \mathbb{U} \end{cases} \tag{5.9}$$

where I is the identity matrix. There exist many efficient methods for obtaining P_i numerically, such as the LMI solvers of *Matlab Robust Control Toolbox*. Next, we show that this analysis is easily extended to include any plant and controller.

5.4.3 Case Study: DC Motor

To evaluate YOLO, we have modeled a closed loop system where the plant, $G(s)$, is a DC motor and the controller, $P(s)$, is a Proportional-Integral-Derivative (PID) controller.

✦✦✦

Problem Formulation

Figure 5.3 gives a simple abstraction of the DC motor. The open-loop transfer function of the DC motor is given in Eq. (5.10), where the rotational speed, $\dot{\theta}$, is the output and the voltage, v, is the input [144].

$$G(s) = \frac{y(s)}{u(s)} = \frac{K}{(Js + b)(Ls + R) + K^2} \tag{5.10}$$

We further simplify the model by assuming that $L << R$ and hence obtain a first order system, as shown in Eq. (5.11).

$$G(s) = \frac{K_o}{T_o s + 1} \tag{5.11}$$

where K_o and T_o are calculated by using Eq. (5.12) and Eq. (5.13), respectively.

$$K_o = \frac{K}{R\,b + K^2} \tag{5.12}$$

$$T_o = \frac{R\,J}{R\,b + K^2} \tag{5.13}$$

The PID controller is a simple yet versatile feedback compensator structure [144]. Its transfer function is shown in Eq. (5.14).

$$P(s) = \frac{u(s)}{e(s)} = K_p + \frac{K_i}{s} + K_d s \tag{5.14}$$

where K_p, K_i, and K_d represent the proportional, the integral, and the derivative gains, respectively.

From Eq. (5.11) and Eq. (5.14), the closed loop transfer functions, $F_{inactive}(s)$ and $F_{active}(s)$, are developed, as shown in Eq. (5.15) and Eq. (5.16).

$$F_{inactive}(s) = \frac{K_o K_d s^2 + K_o K_p s + K_o K_i}{(T_o + K_o K_d)s^2 + (1 + K_o K_p)s + K_o K_i} \tag{5.15}$$

$$F_{active}(s) = \frac{K_o}{T_o s + 1} \tag{5.16}$$

By comparing the aforementioned equations with Eq. (5.2) and Eq. (5.3), and using $n = 2$, the values of the matrices A_i, B_i, C_i, and D_i are obtained as shown below. One extra zero-state has been added to $F_{active}(s)$ in order to make it a second order system as $F_{inactive}(s)$.

$$A_1 = \begin{bmatrix} 0 & 1 \\ \frac{-K_o K_i}{T_o + K_o K_d} & \frac{-(1+K_o K_p)}{T_o + K_o K_d} \end{bmatrix} \qquad B_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$C_1 = \begin{bmatrix} \frac{K_o K_i T_o}{(T_o + K_o K_d)^2} & \frac{K_o (K_p T_o - K_d)}{(T_o + K_o K_d)^2} \end{bmatrix} \qquad D_1 = \frac{K_o K_d}{T_o + K_o K_d}$$

$$A_2 = \begin{bmatrix} \frac{-1}{T_o} & 0 \\ 0 & 0 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad C_2 = \begin{bmatrix} \frac{K_o}{T_o} & 0 \end{bmatrix} \qquad D_2 = 0$$

Evaluation

We assume the following typical set of parameters for the DC motor: $J = 0.01$, $b = 0.1$, $K = 0.01$, $R = 1$. By using the following gains for the PID controller; $Kp = 100$, $Ki = 200$, $Kd = 10$, one obtains the following A_i matrices:

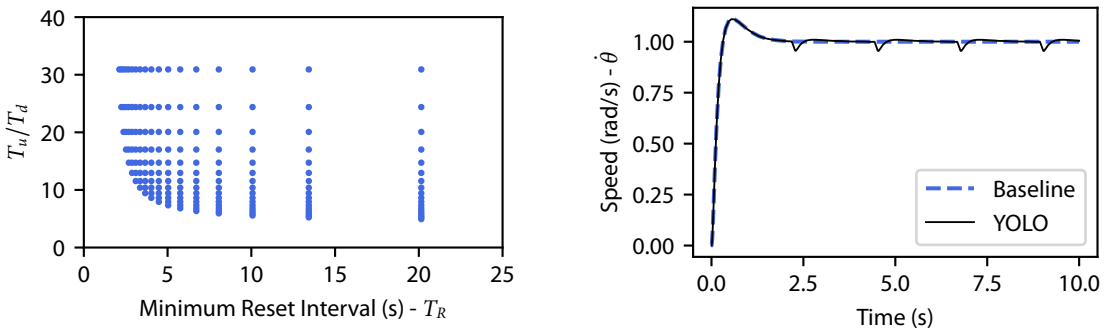
$$A_1 = \begin{bmatrix} 0 & 1.0000 \\ -18.1818 & -10.0009 \end{bmatrix} \qquad A_2 = \begin{bmatrix} -10.01 & 0 \\ 0 & 0 \end{bmatrix}$$

Using basic matrix algebra, one obtains $\lambda(A_1) = \{-7.6125, -2.3884\}$ and $\lambda(A_2) = \{-10.01, 0\}$. From which, $\lambda^- = 2.3884$ and $\lambda^+ = 10.01$ are obtained. We choose $\lambda \in (0, \lambda^-) = 0.1$ and $\lambda^* \in (\lambda, \lambda^-) = 2$. From Eq. (5.8), $a_1 = 0$ and $a_2 = 2.0146$. Hence, a equals 2.0146. Finally, by using Eq. (5.6) and Eq. (5.7), we get $\tau = 1.0603s$ and $T_u/T_d = 30.9202$. This leads to a possible selection of $T_R > 2\tau = 2.15s$ with $T_d = 67ms$.

Additionally, using *Matlab Robust Control Toolbox* to solve Eq. (5.9) would give us the following two positive definite symmetric matrices, which are required to construct Lyapunov stability.

$$P_1 = \begin{bmatrix} 0.4873 & -0.0321 \\ -0.0321 & 0.0479 \end{bmatrix} \qquad P_2 = \begin{bmatrix} 0.0269 & 0 \\ 0 & 0.0551 \end{bmatrix}$$

It is worth noting that choosing different values for the arbitrary scalars λ and λ^* would result in different valid combinations for τ and T_u/T_d. For instance, Figure 5.4a summarizes the allowed minimum reset times (T_R) and the corresponding T_u/T_d. Every point on the graph represents one selection of λ and λ^*. This



(A) The relation between the minimum reset time T_R and its corresponding T_u/T_d for arbitrary selection of λ and λ^*.

(B) The time response of the DC motor rotational speed for a *Baseline* system and YOLO-ized one.

FIGURE 5.4: Simulation results for the DC motor.

flexibility of λ and λ^* is highly important in order to satisfy any hard constraint proposed by the controller (i.e, the controller might require a certain T_d to operate normally after a reset).

Furthermore, we simulate the system shown in Figure 5.2 by using *Matlab Simulink* with the aforementioned parameters for the DC motor and controller. Figure 5.4b shows the output time response for a *Baseline* system versus a YOLO-ized one ($T_R = 2.15s$) and ($T_d = 67ms$). YOLO introduces approximately a 4% drop in engine speed. This result highlights how YOLO can be realized solely relying on the unique CPS properties.

5.5 Experimental Analysis

In order to empirically evaluate YOLO, we study two distinct CPSs: an Engine Control Unit (ECU) and a UAV flight controller (FC)[1]. Each case study provides its own challenges to determining the feasibility of YOLO, because each one is different with respect to the physical component under control and thus the complexity of the controller. The main questions we study are: for which reset periods T_R is the system safe (a) for the ECU and (b) for the FC. Similarly, we also study how the performance of the system is impacted by resets.

5.5.1 Case Study: Engine Control Unit

An ECU controls the combustion process an engine. For a combustion engine to produce the right amount of power, the ECU must inject fuel into the internal chamber, mix it with air, and finally ignite the air-fuel mixture, all at the right timings as shown in Figure 5.5.

[1] Videos for the experiments can be found at: https://miguel.arroyo.me

✦✦✦

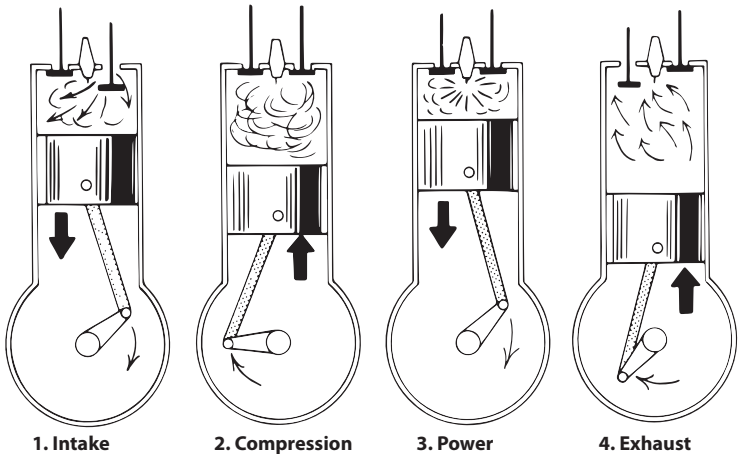


FIGURE 5.5: The four stroke cycle used by combustion engines: (1) intake (2) compression (3) power and (4) exhaust.

HOW IT WORKS There are two rotating parts, the crank and camshaft, inside an engine. The ECU observes their revolutions to determine the state of the engine. The number of input signals that must be observed to correctly determine the engine state depends on the shape of the crank and camshaft. Then, the control algorithm interpolates the time to properly schedule ignition and injection events. Once the ECU has determined the phase of the combustion cycle the engine is in, it will use other measurements from sensors, such as throttle position, temperature, pressure, air-flow and oxygen to accurately determine the air-fuel mixture to be injected.

PLATFORM We use the rusEFI open-source ECU [145] and a Honda CBR600RR engine, a very commonly engine used by FSAE racing enthusiasts. The source-code is written in C/C++ running on top of an open-source real-time library operating system called ChibiOS [146] and is designed to run on a STM32F4-Discovery board, a widely popular micro-control unit (MCU). This board contains a 168MHz ARM Cortex-M4 processor with 192KBytes of SRAM and 1MB of non-volatile flash memory. The MCU contains only a Memory Protection Unit (MPU) with eight protection regions.

RESET MECHANISM Realizing YOLO involves selecting an appropriate reset mechanism. For the ECU, we choose to power cycle the MCU, which effectively clears out all hardware state (i.e., including volatile memory). Simple power cycling, or reboots, provide strong security advantages as it can be triggered externally,

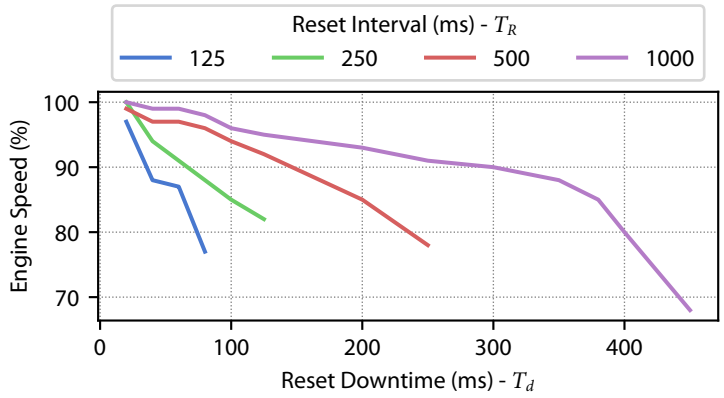


FIGURE 5.6: A sweep of the reset interval T_R and reset downtime T_d to study the effects on engine speed. We observe that for certain combinations of T_R and T_d the engine speed approximates 100%.

without any software. Additionally, it protects against attacks which may freeze the configuration of certain hardware peripherals. Power cycling incurs certain costs, specifically the cost of rebooting the chip and the time for the startup routines to reinitialize the controller. However, we found that the cost of rebooting the chip was on the order of microseconds and thus completely inconsequential compared to the latency of the startup routine. The non-interactive version of rusEFI's startup time is approximately 20*ms*.

DIVERSIFICATION STRATEGY We implement a static variant of Isomeron [147] that provides execution path randomization. Isomeron introduces a hybrid approach that combines code randomization with execution path randomization. Its main objective is to mitigate code-reuse attacks. The high level idea is the following: two copies of the program code are loaded into the same address space and execution is randomly transferred between the two on every function call. The original implementation of Isomeron uses dynamic binary instrumentation techniques which are not feasible on resource constrained devices. We leverage a binary rewriter to implement a static version of Isomeron which makes the technique suitable for our targeted resource constrained devices.

✦✦✦

EVALUATION

The synchronization time for the ECU is dependent on the number of engine cycles that must be observed. The ECU must observe two engine cycles to determine whether it is synchronized with the engine's rotation.

Additionally, it must observe enough engine cycles to compute properties that must be integrated over time (eg. acceleration requires three engine cycles). Assuming an engine speed of 4500RPM (i.e., approx 75Hz), each engine cycle takes 13ms, where synchronization takes 39ms. Combining the previous software timing constraints along with the physical safety constraints, we set a range of T_R and T_d to perform our experiments according to Eq. (5.7).

Resets Figure 5.6 shows the change in engine speed—for an unloaded engine[2]—as a percentage for the sweep. Maintaining the engine speed can be satisfied for a wide range of T_R and T_d where the engine speed is approximately 100%. We note what happens when the engine speed drops significantly. At some point, the ignition & injection steps are unable to generate enough energy to overcome friction, and the engine comes to a stop. We refer to the specific engine speed at which this failure occurs as the stopping threshold. As we reset more frequently, we observe lower engine speeds without crossing the stopping threshold during operation as we stay within the system's safe region. We also note that the actual stall threshold varies non-linearly with T_R and T_d, most likely due to environmental factors and the large variability in the internal combustion process. Under engine load, we expect the stall threshold to vary less due to the added inertia. For our system, we can therefore conclude that there are specific combinations of T_R and T_d for which stability can be satisfied even as the system misses events.

Diversification For the ECU we studied the effects of our Isomeron implementation. The results showed our version introduced a constant slow down of approximately 2.13x, primarily due to its use of a hardware random number generator used to implement the Isomeron diversifier that switches between functions. While this slow down may seem large, the original application had more than sufficient slack to accommodate it. To put this into perspective, even with the slow down, our ECU was still within typical timing accuracy of commercial systems ($2 - 3°$ delay). To further test the limits of our system, we swept ignition and injection delays up to 8x, but saw no observable effects on RPM for our engine.

5.5.2 Case Study: Flight Controller

The flight controller is designed to ensure the stability and control of an aircraft. It does so by controlling translation along the x, y, z directions and rotation about the x, y, z axes (i.e, the attitude).

[2]No access to a dynamometer was available.

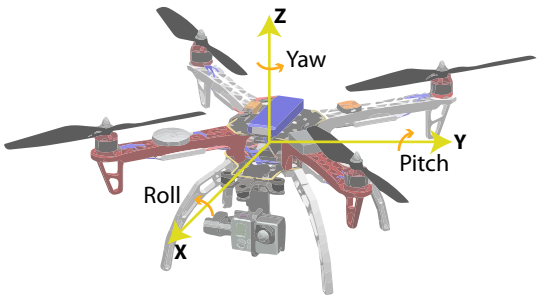


FIGURE 5.7: Axes to control a quadcopter's attitude (i.e., pitch, roll, and yaw) about its center of gravity.

✦✦✦

HOW IT WORKS The flight controller is primarily responsible for ensuring attitude stability while aiding a pilot or performing autonomous flight. It must read all of the sensor data and filter the noise in order to calculate proper output commands to send to its actuators. In particular, we focus on quadrotor helicopters more commonly referred to as quadcopters. Controlling these quadcopters involves operating four independent rotors to the provide six degrees of freedom. Sensors measuring a number of physical properties are then fused together to estimate the position and attitude of the quadcopter. This estimation, similar to the ECU, requires a certain number of measurement samples to be observed before output is produced. This output is then used by other components that determine the best command actions for the system.

PLATFORM We use the PX4 open-source FC [148] with a DJI F450 quadcopter air-frame, a very common DIY kit favored by enthusiasts. The PX4 FC provides attitude and position control using a series of sensors, such as GPS, optical flow, accelerometer, gyroscope, and barometer. The PX4 controller software includes a variety of flight modes ranging from manual, assisted, to fully autonomous. The source-code is written in C/C++ and supports multiple kinds of OS and hardware targets. Specifically, we use the Pixhawk board based on the same series of MCU as that used in the ECU case study.

RESET MECHANISM Similar to the ECU case study, we first attempted simple reboots. The downtime T_d for PX4 was found to be around 1.5s. This higher reset time in comparison to the ECU was not unexpected due to the higher complexity of the quadcopter controller. Given the more sensitive physical dynamics of the quadcopter, simple rebooting is not effective, i.e., the quadcopter crashes very often, prompting the need of a more efficient approach. This led us to explore alternate, more performant reset mechanisms. We found that much of the startup time was spent in initializing data structures and setting up the system for opera-

tion. So, we create a snapshot of RAM after all the initialization is complete and use it to start the system in approximately 3*ms*. This snapshot can additionally be verified and signed for increased security.

The snapshot is stored in a special region of flash and at the following boot, the saved state is restored. The special flash region is protected, and locked by the MPU. This provides a consistent restoration point for the system's lifetime. Snapshot-ing was implemented as an extension to the NuttX library OS [149] used by the Pixhawk PX4 target. The 3*ms* that the snapshot restoration takes is primarily dominated by the time required to write data from flash to RAM.

When the snapshot is taken, and what data is stored in the snapshot, has implications on the capabilities of the system. Therefore, we allow the designer to annotate data regions to be persisted and placed them in a memory location excluded from the snapshot & restoration mechanism. The annotations are accomplished via source code compiler directives and linker modifications.

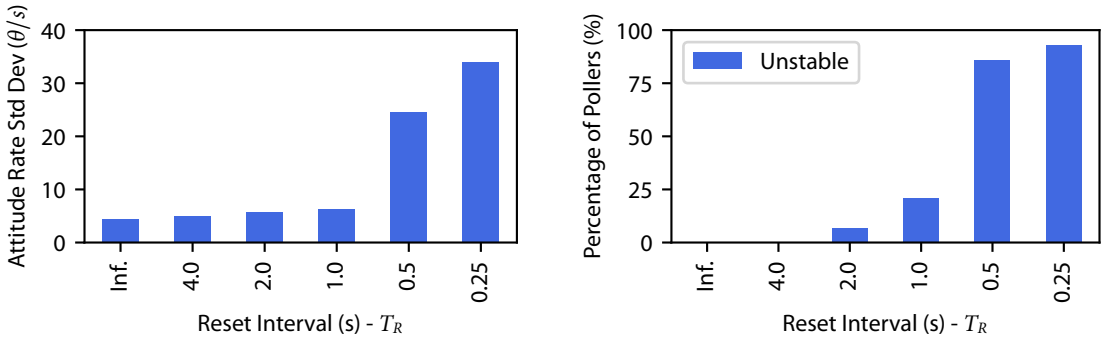
Depending on the flight mode for the quadcopter, the snapshot has different requirements as to what data can be reset and persisted. For the autonomous flight mode for example, coordinates for the quadcopter's flight path could appropriately be made part of the snapshot taking care to use absolute coordinates were possible. However, including the flight path in the snapshot would prevent the quadcopter's path from being modified mid-flight. If this capability is desired, the data would need to be persisted across resets and protected in some way as discussed in § 5.3. The assisted flight mode has fewer limitations. For the assisted flight mode, which only requires the pilot inputs, a simple snapshot of RAM taken after the sensors have been calibrated is sufficient, as the system can recover the state that it needs by re-observing the environment. For optimal security, the snapshot could be taken once in a controlled and secure environment, as long as, the system was initialized with the correct parameters.

Diversification Strategy Here, we chose to implement a simple variation of conventional stack canaries. Canaries, or integers placed in memory just before the stack return pointer, are useful against stack overflows. On each reset, we randomly generate new canaries used by the program.

✦✦✦

Evaluation

In this section, we discuss the results of our experiments on the flight controller platform.



(A) The effects of resets on the quadcopter stability for various reset intervals. These results, quantify our observations from the poll conducted.

(B) The results of the poll conducted to determine at which reset interval (T_R) the quadcopter will start to become unstable during hover. From this we can determine a minimum T_R for the system.

FIGURE 5.8: Experimental evaluation for quadcopter case study.

RESETS To quantify the effects of resets, the standard deviation of the quadcopter's attitude rate over time was used. The results are shown in Figure 5.8a. The results show little impact on the attitude for $T_R > 1$s and a large spike for smaller values; this indicates that for $T_R \geq 1$s the stability of the system is roughly equivalent to the system without YOLO. At lower T_R periods, we see a large spike in the standard deviations, which correspond to when we observe the system to start oscillating resulting in decreased controllability.

To better gauge the threshold at which a pilot would begin to detect these oscillations, or in other words, the lower limit for T_R, we conducted a survey among a set of 20 students. The survey was conducted using an ABX test methodology where various videos of the quadcopter with YOLO during flight for different T_R were shown. Before conducting the survey, users were shown an example video of a stable and unstable flight. They were then shown videos in a random sequence and asked to determine whether there were any observable oscillations during hover flight. The results are shown in Figure 5.8b indicating that oscillations become significantly observable somewhere between T_R of 0.5 to 1 second.

DIVERSIFICATION Similar to the ECU, we sought to observe any difference in the system's behavior introduced by diversification. Our simple strategy in the case of the quadcopter, had a negligible effect. This is not surprising as stack canaries have significantly low overheads. To further stress the effects of delays on our system, we effectively lowered the control loop rate by $1.5x$ and $2x$ which resulted in approximately a 7.6% and 11.3% change in the attitude standard deviation, respectively. In normal flight, we noticed no visual sign

of these effects or any noticeable difference in maneuverability with respect to pilot inputs in non acrobatic meaneuvers (e.g., waypoint based missions).

5.6 Limitations

Temporary loss of control In some CPSs even a temporary loss of control due to resets may be unacceptable. These cases have to be carefully evaluated. If temporary loss of control is completely unacceptable, then multiple replicas are necessary to provide fail safe operation. In this situation, interleaved resets can be used to implement YOLO to enhance security of the ensemble system. The practicality of this approach is validated by its use in the Boeing 787 which recently deployed interleaved resets [150] in response to a software error that triggered a simultaneous reset of all flight control systems (a safety issue, not a security problem).

Multiple Interacting Components Our case studies look at systems with single computing components. Larger and more complex CPSs may have multiple interacting components in which timing and communication challenges may arise. In order to realize YOLO on such systems, one possible solution is to think of each component as a microservice and follow a reset strategy similar to that proposed by Candea *et al.* [151] which involves recursively rebooting services depending on the time slack available to them. Given that the timings of interactions between components is part of the design, this information can be leveraged to implement YOLO.

Persisting Data It is possible that some CPSs must carry over data across resets or even persist it to non-volatile memory. One scenario in which this arises is logging. Given the periodicity of resets, this may be difficult since there may not be enough time to allow for writes. To handle this limitation and capture data for evaluation for example, we architect the system to include an additional logging device that listens to sensor data. This logging device is not in any way tied to the control of the system. Other approaches may be possible depending on the reset timing.

Wear & Tear YOLO may have miscellaneous effects on CPSs that may not have been observable during our evaluation. These effects can include additional wear and tear of components for example. It is difficult to say whether YOLO may have long term effects. However, after having explored these systems, the physical

components are built with ample tolerance margins such that we may never see any effects for the duration of the system's lifetime. For example, the rotors on quadcopters are Brushless DC motors (BLDC). BLDCs rely on electronic commutation as opposed to mechanical commutation and therefore do not suffer from much wear and tear.

5.7 Related Work

Software Rejuvenation Software rejuvenation was introduced by Huang *et al.* [152] to address the issue of *heisenbugs*, non-deterministic failures that can occur as the execution of a program "ages" [153, 154]. The concept of software rejuvenation has been previously proposed for security in general-purpose systems [155, 156]. As in YOLO, software is reset intermittently from a clean state, but with the hope that this will increase the time until another heisenbug occurs. In contrast to software rejuvenation, where reset timings are determined by failure rates, the frequency of YOLO's resets are determined by the length of time the system can remain safe once security is possibly compromised. In other words, YOLO's resets should aim to be as quick as possible while still ensuring correct operation to achieve maximum security.

Closely related to YOLO is work by Abdi *et al.* [157]. Concurrently with our work, Abdi *et al.* [157] proposed a reset based approach to secure CPSs. In contrast to YOLO, their design requires additional hardware components for maintaining the safety of the system during resets. This may make it more difficult to retrofit into legacy systems, increases cost, and requires a larger trusted computing base (TCB). YOLO has no such requirement. Most recently, Romagnoli *et al.* [158] considers software rejuvenation design from a control theoretic perspective extending concepts from both YOLO and [157].

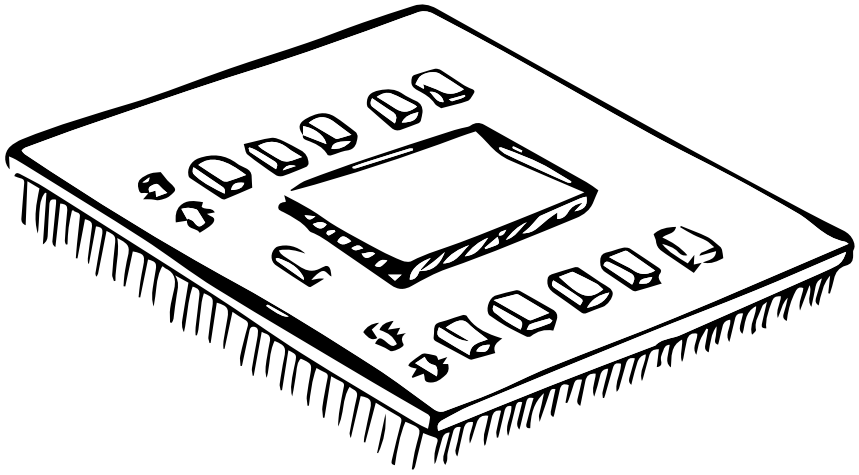
Proactive Recovery Research on byzantine faults has focused on techniques meant to tolerate benign faults (e.g., hardware errors such as a hard drive failing, etc). The assumption of benign faults, however, is not valid in the presence of a malicious adversary. Proactive recovery extends the traditional notion of Byzantine faults to provide a solution to malicious attacks, operator mistakes, and software errors. Castro and Liskov [159] propose BFT, a proactive recovery mechanism that recovers replicas periodically even if there is no reason to suspect that they are faulty. Like YOLO, the proactive recovery ensures a bounded time horizon over the which the attacker can affect the system; in the case of Byzantine faults, this also limits the amount of replicas that can be compromised at any given time. Unlike YOLO, BFT requires replicated hardware, a

much longer recovery period (e.g., in the order of minutes in contrast to YOLO's milliseconds), and does not integrate diversification to thwart multiple attempts by an adversary. More recently, Mertoguno *et al.* [160] have observed that unique physical properties of CPS can be used along with byzantine fault techniques to improve security. They devise a system that takes advantage of the inherent redundancy available in CPS along with inertia to design a Byzantine Fault Tolerant system.

Checkpointing Checkpointing has been widely used in the context of fault tolerant systems, allowing a system to roll back to a previously valid state in the event of failure [161, 162]. Checkpointing has been extended for security usecases by encrypting memory snapshots to prevent leaks [163]. However, checkpointing in general suffers from drawbacks with regards to verifying the integrity of snapshots. Unlike traditional checkpointing, YOLO always resets to a known secure state which ensures the integrity of the snapshot, relying on CPS properties to recreate discarded information. Moreover, because YOLO does not need to continuously checkpoint memory, it does not incur the additional overheads of synchronization to ensure snapshots are consistent.

Part III

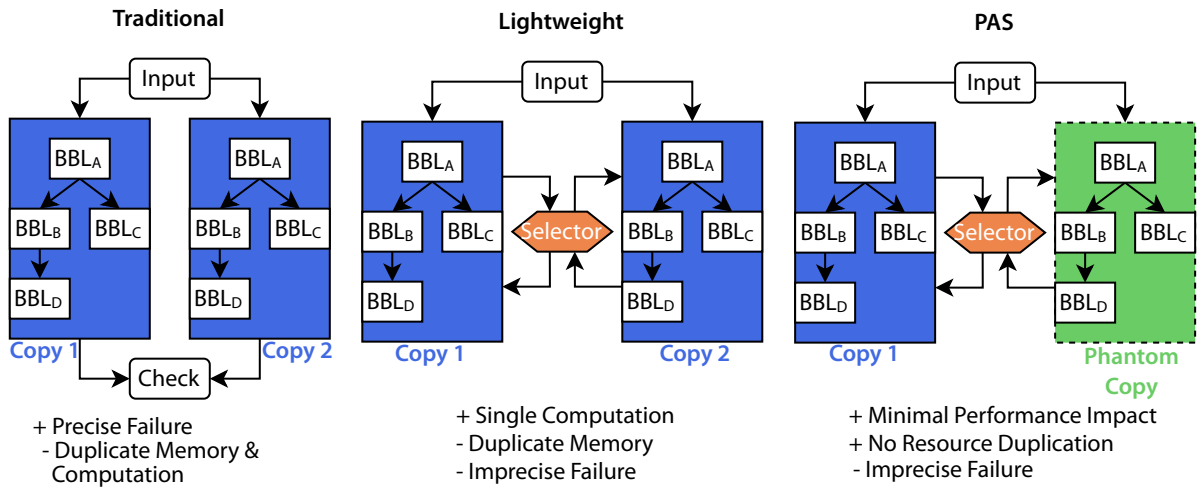
Revisiting Age-old Computing Abstractions for Efficient Security



6 Phantom Address Space (PAS)

YOLO may not be applicable for all CPSs, whether it be due to limited inertia, or complex control algorithms that require persistent data. As result, CPSs need additional mechanisms meant for operating under stricter requirements that can still provide robust security. The defense presented in this chapter provides a standalone solution that can not only be layered with YOLO, but is applicable for a broad spectrum of resource constrained devices.

6.1 Overview



(A) Traditional N-variant systems execute two program copies in different processes.

(B) Lightweight N-variant systems (e.g., Isomeron [147]) randomly select one program copy to execute via a selector component.

(C) Phantom Address Space systems operate on two program copies, where only one of them is stored in physical memory and the other (virtual copy) is the phantom.

Figure 6.1: Comparison between different N-Variant Execution approaches.

One approach to mitigating exploits that rely on memory corruption vulnerabilities is to employ *artificial software diversity* [135], as a means to increase the cost (or decrease the feasibility) of mounting successful

attacks. N-version/variant programming is a classic technique, where multiple variants of a program are executed together to improve the security and reliability of a system [164, 165]. Figure 6.1(a) shows an example of a 2-variant system that executes two different copies of the program, at the same time, with the same set of inputs. The outputs of the two copies are compared by a *checker module*, which raises an exception once a discrepancy is detected. Although N-variant execution provides a precise failure model in case of attack/error detection, it introduces significant memory and runtime overheads [166]. One way to optimize the N-variant systems is to add a selector component that executes only one copy of the program at runtime [147], as shown in Figure 6.1(b). This optimization reduces the computational workload by compromising with an imprecise failure model. However, the lightweight variant still suffers from the same memory overheads as the basic 2-variant one.

We introduce a novel concept, called *Phantom Address Space* (PAS), which can be used to eliminate most of the memory and performance overheads of N-version/variant execution on 32 and 64-bit systems. The main idea of PAS is to use modifications to the memory addressing functions, in order to generate a *phantom copies* of the program (see Figure 6.1(c)). These phantom copies does not exist in physical memory, and, therefore, the associated memory overheads of multiple variants are mostly eliminated. A *selector component* orchestrates the control-flow of the program between the two "virtual" copies, effectively resulting in only one copy being executed at any given time. This approach reduces the runtime overhead close to vanilla execution. PAS is realized entirely in microarchitecture and as a result can be deployed with legacy software.

We use PAS to implement a diversification protocol known as Execution Path Randomization (EPR) that makes it very likely for the program to crash in the event of an attack. Isomeron, a software-only implementation of EPR was first introduced by Davi *et al.* [147]. This technique represents a 2-variant system to thwart the attacker's ability to guess the exact locations of desired gadgets used in code-reuse attacks (CRAs) a priori [139]. PAS achieves the same security goal by ensuring that the locations (i.e., addresses) of program instructions in the phantom copy are shifted by Δ and δ bytes (compared to the location of the same instructions in the original program copy). PAS then switches between the phantom copies and the original. On every fetch, the hardware randomly decides from which phatom it should fetch an instruction from. Since the attacker is not privy to the random guesses, with high probability they cannot carry out a CRA. In other words, if the attacker supplies an incorrect address that does not follow the diversification protocol, the exploited program will crash. PAS can provide up to N addresses per instruction at any given time, meaning if the attacker has to reuse P instruction sequences to complete an attack, the probability of detecting the attack

is $1 - (1/N)^P$ without any false positives. For example, for $N = 256$ (as in our current prototype) and $P = 5$, then the probability of an attack succeeding is 1 in 1 trillion. This kind of protection makes this technique suitable to be used as a standalone solution, or in tandem with other, heavier-weight hardening mechanisms.

A naive implementation of PAS would require each instruction to be stored in N locations to have N addresses. Consequently, the capacity of all PC-indexed microarchitectural structures would be halved by N, heavily impacting performance. Further, this requires changes to the compiler, linker, loader, *etc.* PAS avoids these problems by intentionally *aliasing* the different instruction addresses so they point to the same instruction, allowing us to serve the N instructions from one copy. This idea is similar to how multiple virtual addresses can point to the same physical addresses (used to implement copy-on-write [167]) with two key differences: first, in PAS the N phantom addresses correspond to the same virtual address, not a physical address; and second, the PAS phantom addresses do not need to be page-aligned as required for data synonyms—i.e., PAS phantom address can be arbitrarily offset. The first difference ensures that PAS can be handled at the application level without requiring significant changes to the operating system (OS), which manages the virtual-to-physical address mappings, while the second is key to providing security.

6.2 Threat Model

We consider a threat model that is consistent with previous work on code-reuse attacks and mitigations [147, 168, 169, 170]. We assume that the adversary is aware of the applied defenses and has access to the source code of the target program. Furthermore, we also assume the target program suffers from memory safety-related vulnerabilities that allow the adversary to read from, and write to, arbitrary memory addresses. The attacker's objective is to (ab)use memory corruption and disclosure bugs, mount a code reuse attack, and achieve privilege escalation.

Hardening Assumptions. We assume that the underlying OS enables both ASLR and W^X protection—i.e., no code injection is allowed (non-executable data), and all code sections are non-writable (immutable code). Thus, attacks that modify program code at runtime, such as rowhammer [171], are out of scope. We also do not consider non-control data attacks [131], such as Data-Oriented Programming [172] and Block-Oriented Programming [173]. This class of attacks only tamper-with memory load and store operations, without inducing any unintended control flows in the program. This limitation also applies to prior work

as well [147, 168, 170, 174]. Lastly, every other standard hardening feature (e.g., stack-smashing protection [175], CFI [174]) is *orthogonal* to PAS; our proposed scheme does not require nor preclude any such feature.

6.3 Framework

PAS consists of N *phantoms* (domains). It requires every instruction in the program to have N unique addresses. To assign the addresses, we use a mapping function, $addr_p = f(va, p)$, which takes the instruction virtual address, va, and a phantom index, p as inputs and returns the phantom address, $addr_p$. With this function any instruction is mapped by f to a unique location in each of the N phantoms. The function f does not have to be kept a secret, as security is purely derived from the random selection of p during a fetch. For mapping a phantom address to its original virtual address, we use the inverse function, $va = f^{-1}(addr_p)$. To enable the inverse function, we ensure that the phantom address encodes the phantom index p.

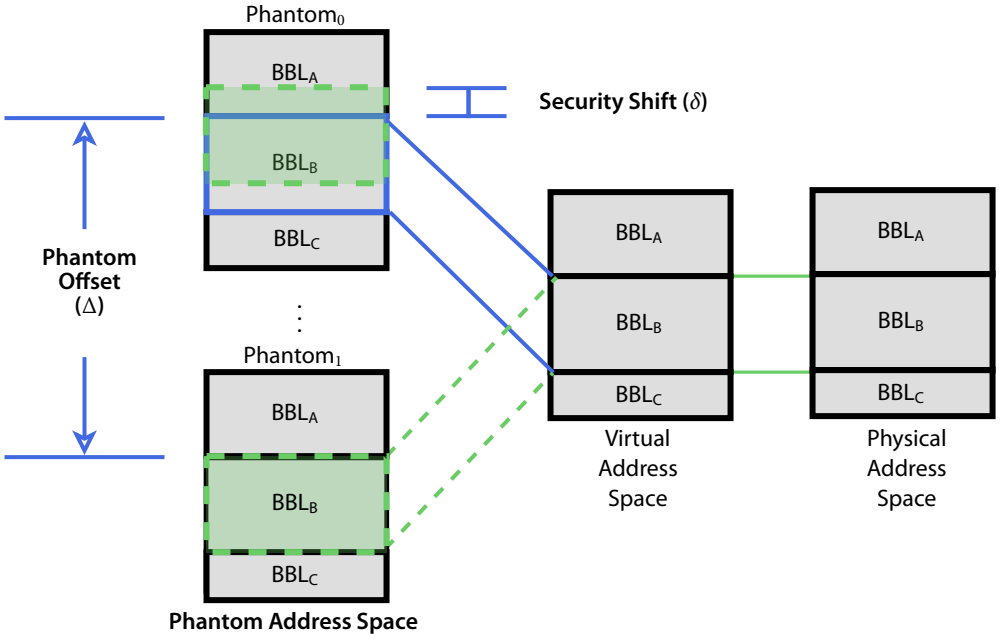


Figure 6.2: The virtual to physical basic block mapping for PAS. BBLs are only duplicated in virtual address space.

There are four main operations to realize PAS: *Populate*, *Randomize*, *Resolve*, and *Conceal*.

Populate PAS creates multiple phantoms of basic blocks, and populates them in the phantom address space. The left-hand side of Figure 6.2 shows a program with two Phantoms, such that every basic block (BBL) has two different addresses in Phantom_0 (aka the original domain) and Phantom_1. PAS separates the

two Phantoms by a phantom offset, Δ, in the phantom space. To add discrepancy between the Phantom copies, we introduce a minor security shift, δ, so that they are not perfectly overlapped after removing Δ. This is shown by the shaded basic block in Figure 6.2 and is necessary for security, as we will discuss in more detail in § 6.6. The inverse mapping function f^{-1} maps all phantoms to a single name in the virtual address space, which is then translated to a physical address by the OS.

Randomize We modify the hardware to randomize program execution between the Phantoms at runtime. For example, some basic blocks will be executed from Original (Phantom_0) while other basic blocks will be executed from any other Phantom. Correctness is unaltered because all Phantoms provide the same functionality by construction.

Resolve Accessing different instruction addresses at runtime incurs additional performance overheads as each addresses needs to be translated to a virtual address and then a physical one before usage. To mitigate this problem, PAS uses the inverse mapping function f^{-1} to resolve the different Phantoms to their archetype basic block. By doing so, the processor back-end continues to operate as if there is only one copy of the program in the phantom address space.

Conceal Normal programs push return addresses to the architectural stack to help return from non-leaf function calls. The attacker may learn the domain of execution, the Phantom index, by monitoring the stack contents at runtime using arbitrary memory disclosure vulnerabilities [139]. Thus, to preserve name confusion, we need to conceal the execution domain of the instructions.

6.4 Construction

In this section, we discuss alternative design choices for the different operations in the PAS framework.

Populate Many approaches can be used to populate the Phantoms. One approach is to use the most significant bits (MSBs) to separate the program copies in the phantom space. For example, a Δ of `0x8000_0000_0000_0000` will create two phantoms on 64-bit systems, where each phantom resides in one half of the address space. This approach is acceptable for 64-bit systems because `VA` allows for 64 bits, yet only 48 are used in practice, leaving the higher order bits available for phantom addresses. However, this is costly

for 32-bit systems as it will reduce the effective range of addresses a program can use by half. Instead, to store the phantom index we add n additional bits to the hardware program counter, while maintaining the 32-bit virtual address space of the program. This allows PAS to generate $N = 2^n$ phantoms. Specifically, f, sets the additional n bits at control-flow transitions to randomize the execution at runtime. For simplicity, we set the phantom offset as $\Delta = 1 \ll 32$ and the minor security shift of any phantom to be a multiple of the phantom index (i.e., $\delta_p = p \times \delta$).

RANDOMIZE PAS can randomize program addresses at any level of granularity, ranging from individual instructions to entire programs. In the rest of this chapter, we use basic blocks as our elements of interest. We do not evaluate finer granularities here due to the lack of a strong security need. We define the basic block as a single entry, single exit region of code. Thus, any instruction that changes the `PC` register (referred to by control-flow instructions, such as `jmp, call, ret`) terminates a BBL and starts a new one.[1]

CONCEAL We can prevent attackers from learning the execution domain in a number of ways. One straightforward way is to encrypt the return address with a secret key and only decrypt it upon function return. Another key-less, and low overhead, method that we implement is to split this information so that the public part is what is common between the phantom domains, and the private part that distinguishes the domains is hidden away without architectural access.

We split the return addresses between the architectural stack and a new hardware structure called the Secret Domain Stack (SDS), which by construction is immutable to external writes. SDS achieves this goal by splitting the return address $(32+n)$ bits into two parts; the n-bits, which represent the *phantom index* (p), and the lower 32 bits of the address, which encodes the *security shift* (δ). With each function `call` instruction, the lower 32 bits of the return address are pushed to the architectural (software) stack, whereas the phantom index p is pushed onto the SDS. A `ret` instruction pops the most recent p from the top of SDS and concatenates it with the return address stored on the architectural stack in memory. While under attack, the return address on the architectural stack will be corrupted by the attacker. However, the attacker cannot access SDS so they cannot reliably adjust the malicious return address to correctly encode δ, leading to an incorrect target address after PAS merges the malicious return address with the phantom index p from SDS. Deployment issues with the SDS such as sizing, overflows, multithreading, *etc.* are described in § 6.9.

[1] Some compilers, such as LLVM, deviate from this definition and treat `call` instructions as part of the BBL.

6.5 Correctness

Since the addresses are selected during a fetch, how can we be assured that all PC relative computations are used in the correct way as encoded in the original program? In order to discuss the correctness of PAS's construction and operation, we consider the structured programming theorem [176], in which a program is composed from any subset of the control structures: *Sequence*, *Selection*, *Iteration*, and *Recursion*. We show that PAS does not affect the four structures. For simplicity, let us assume $n = 1$, so that only two phantoms exist, Original or Phantom. First, the *Sequence* structure represents a series of ordered statements or subroutines executed in sequence. PAS guarantees this property by executing the statements (instructions) of a BBL in the same domain of execution (either Original or Phantom).

For handling the *Selection* structure, let us assume that a program is represented by a binary tree with a branching factor of 2. Hence, we define two types of such trees. Type `I` is the tree where nodes are represented by the BBLs of *committed* instructions. Type `II` is the tree where each node has the address of the first instruction in the *executed* BBL. The edges are given by the direction taken by the last instruction of each BBL. The root of the tree is the first instruction fetched from the `_start()` section of a binary (or the address of this instruction in the address based tree). The leaf node is the last BBL of the program (or the address of this BBL in trees of Type `II`). In the case of PAS, every taken branch on the tree of Type `I` is the same as every taken branch on the tree of Type `II`, i.e., program functional decisions are not affected. However, the contents of the tree nodes in Type `II` trees would be different for each program execution, as each BBL will be fetched from either Original or Phantom domain and addresses of these domains differ by Δ. In other words, after the branch is resolved to be taken or not taken, PAS operates on the outcome and randomly chooses the domain of execution for the next basic block.

The above argument also applies for the *Iteration* control-structure, in which the same basic block is executed multiple times. PAS does not change the functionality of the basic block, however, each time the basic block is executed it will be executed in one of the phantom domains. The *Recursion* construct is similar to iterative loops and thus is guaranteed to be executed correctly with PAS. The same proof holds for $n > 1$ by making the branching factor of the tree equals 2^n. From the user perspective, the program will produce the same result, since the order and flow of instructions has not changed (Type `I` trees are unique). However, from the perspective of an entity that observes only addresses, each execution of a program appears to be a

different sequence of addresses since the addresses of individual basic blocks will be altered (Type `II` trees are not unique). This enables a substantial security benefit as we show next.

6.6 Code Reuse Protection

Code reuse attacks (CRAs) are a widely popular class of exploit techniques that, as the name implies, reuse existing application code as their payload. This allows adversaries to overcome the limitations of previous techniques such as code-injection in light of W^X. The most popular class of CRAs, are attacks that chain together gadgets whose last instruction is a `ret` are known as return oriented programming (ROP) attacks [177, 178]. To mount a ROP attack, the attacker has to first analyze the code to identify the respective gadgets, which are sequences of instructions in the victim program (including any linked-in libraries) that end with a return. Second, the attacker uses a memory corruption vulnerability to inject a sequence of return addresses corresponding to a sequence of gadgets. When the function returns, it returns to the location of the first gadget. As that gadget terminates with a return, the return address is that of the next gadget, and so on. As ROP executes legitimate instructions belonging to the program, it is not prevented by W^X [179]. Note that variants of ROP that use indirect `jmp` or `call` instructions, instead of `ret`, to chain the execution of small instruction sequences together also exist, dubbed jump-oriented programming (JOP) [180] and call-oriented programming (COP) [181], respectively.

Standard mitigation techniques for CRAs suffer from deployability issues due to the constrained architectures of embedded devices. For example, ASLR which is a widely adopted defense for CRAs is much less effective on 32-bit architectures [130]. Recently, ARM introduced PAC in `Armv8.3A`, which is implemented in the Apple's iPhone XS SoC [182]. The idea is based on a concept known as cryptographic control-flow integrity (CCFI) [168]. For every code pointer, such as return addresses and function pointers, CCFI stores a cryptographically-secure authentication code in the pointer's unused most significant bits. Checking the authentication code of a pointer before any indirect branches prevents control-flow hijacking because the attacker cannot compute a valid authentication code without access to keys. As we will show in § 6.8, to achieve low overheads with this scheme, it is essential to have 64-bit architecture and to apply the solution to only a subset of the pointers: full-application of the idea on a 32-bit processor results in 91% overhead for SPEC2017 (see Figure 6.6). In contrast, PAS aims to enable security for 32 and 64-bit systems, as non-64-bit systems are widely used in CPS and IoT. Thus, there is a need for new low overhead deployable solutions.

✦✦✦

PAS mitigates ROP by *ensuring that the addresses of the ROP gadgets in the gadget chain change after the chain is built*. This will result in undefined behavior of the payload (likely leading to a program crash). Consider the example in Figure 6.2: PAS simultaneously populates multiple (apparent) phantoms of the program code in the phantom address space; to successfully thwart the ROP gadget chain, the location of the ROP gadgets in all phantoms should be different [147].

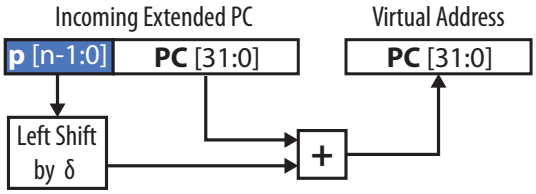
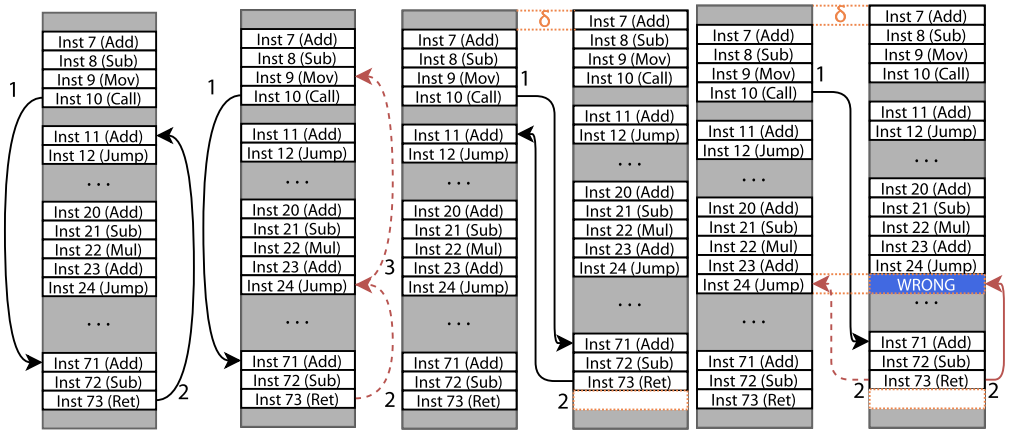


FIGURE 6.3: Mapping the extended PC (i.e., the phantom address space) to the virtual address before indexing into the microarchitectural structures.

Traditional in-place randomization techniques [183, 184] can be used to generate Phantoms. However, using an aggressive randomization approach will complicate the inverse mapping function, f^{-1}, which is responsible for recovering the archetype basic block from the different Phantoms. This will cause performance overheads with almost no additional security (beyond changing the gadget addresses in the phantom copies). PAS adopts a more efficient code layout randomization technique by introducing a security shift, δ, between the individual Phantoms, so that they are not perfectly overlapped after removing the phantom offset, Δ. This simplifies f^{-1} computations (as shown in Figure 6.3) and maintains code locality.



(A) Regular Execution (B) Execution with CRA (C) Execution with PAS (D) Execution with PAS and CRA

FIGURE 6.4: PAS Execution Comparison Scenarios.

While the program is executing, PAS randomly decides which copy of the program should be executed next. Figure 6.4a shows the normal execution of a program, where `Inst 10` changes the control-flow of the program to a different BBL (starting with `Inst 71`). After the called BBL is executed, the control-flow is transmitted to the original landing point (`Inst 11` via a `ret` instruction). Figure 6.4b shows a successful CRA via ROP, in which the attacker uses a memory safety vulnerability to overwrite the return address stored on the stack and divert the control flow to `Inst 24` upon executing the `ret` instruction. Figure 6.4c shows the diversified execution of a program with PAS. For simplicity, we only show two phantoms and use a security shift, δ, sized to one instruction. Each control flow instruction can arbitrary choose to change the execution domain or not. Here, the `Randomize` operation decides to execute `Inst 71` from the Phantom domain. As the attacker cannot predict this runtime decision in advance, they provide the wrong gadget address on the stack (now shifted by δ). Thus, the atacker will end-up executing a `WRONG` instruction, as shown in Figure 6.4d. This `WRONG` instruction may belong to a different BBL or divert the execution to a new undesired BBL. In general, if the attacker makes the wrong guess, they will execute one less (or one more) instruction compared to the desired gadget. If δ is smaller than the instruction size, the attacker will skip a portion of the instruction resulting in an incorrect instruction decoding.

6.6.1 TRAP Instructions

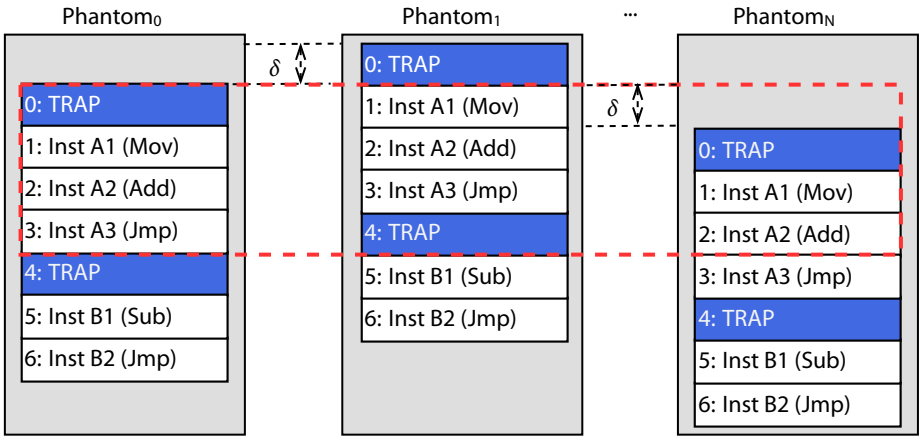


Figure 6.5: PAS TRAP Instruction Precise Failure

To further limit the attack surface of PAS, we add `TRAP` instructions. These instructions are inserted at the beginning of every basic block. With PAS enabled, the security shift, δ, will cause the `TRAP` instruction that exists in the beginning of a BBL in Original domain to appear at different locations of the same BBL in each

of the Phantom domains, as shown in Figure 6.5. The TRAP instruction provides the ability to catch attackers that guess the incorrect diversification boundary while targeting BBLs.

Let us consider the example of Figure 6.4d. Here, the attacker tries to divert the control flow to `Inst 24` upon executing the `ret` instruction. Our `Randomize` operation decides to execute `Inst 71` from the Phantom domain. As a result, the attacker will step on the instruction that follows `Inst 24` in Phantom, which will be a TRAP instruction in the new model. Executing a TRAP instruction results in a security exception and program termination, effectively thwarting the attacker. Under normal conditions (i.e., no attack), programs never execute TRAP instructions as there exists no control-flow transfers to them.[2]

6.6.2 Lightweight Pointer Encryption

Besides ROP, CRA variants also extensively rely on pointer corruption (e.g, JOP/COOP [169, 180]) to subvert a program's intended control flow. There also exist many software-based mitigations for JOP/COOP-like attacks [185, 186, 187, 188]. In this work, we use a hardware-based technique for hardening PAS against them. Since the attacker needs to overwrite legitimate pointers used by indirect branches to launch the attack, we encrypt the contents of the pointer upon creation and only decrypt it upon usage (at a call site). Consequently, attackers cannot correctly overwrite it.

To achieve the above goal our Lightweight Pointer Encryption (PtrEnc) scheme adds two new instructions: ENCP and DECP. The two instructions can either be emitted by the compiler (if re-compiling the program is possible) or inserted by a binary rewriter.

- Encrypt Pointer. (`ENCP RegX`) The mnemonic ENCP indicates an encryption instruction. RegX is the register containing the pointer, e.g., virtual function pointers. The register that holds the encryption key is hardware-based and never appears in the program binary.

- Decrypt Pointer. (`DECP RegX`) The mnemonic DECP indicates a decryption instruction. RegX is the register containing the pointer. The register that holds the decryption key is hardware-based and does not appear in the program binary. As a result, the attacker cannot directly leak the key's value. Moreover, the attacker cannot simply use the new instructions as signing gadgets to encrypt/decrypt arbitrary pointers as they will have to hijack the control flow of the program first.

[2] We modify the hardware to handle the case of a fall-through BBL to prevent legitimately stepping on a TRAP instruction.

Unlike prior pointer encryption solutions, which use weak XOR-based encryption [189, 190], PAS relies on strong cryptography (The QARMA Block Cipher Family [191]). In contrast to full CCFI solutions [168, 182], which use pointer authentication to protect all code pointers including return addresses, our approach only guards pointer usages (loads and stores). Return addresses are handled by PAS randomization, reducing the overall performance overheads, as will be shown in § 6.8.

6.7 Security Discussion

In this section, we present our evaluation results regarding the effectiveness and security guarantees of PAS against CRAs.

6.7.1 Secrets

There are no secret parameters in the basic PAS scheme. The number of phantoms and the security shift can be made public as security comes from the random selection of instruction names (i.e., phantoms). For PAS extensions, a per-process key (used for encryption) should be kept secret for the lifetime of the respective process.

6.7.2 Quantitative Evaluation

ROP-Gadget Chain Evaluation. To evaluate PAS against real-world ROP attacks we use Ropper [192], a tool that can find gadgets and build ROP chains for a given binary. A common ROP attack is to target the `execve` function with `/bin/sh` as an input to launch a shell. As the chain-creation functionality in Ropper is only available for `x86` [192], we analyze SPEC2017 `x86` binaries for this particular exploit and report the number of available gadget chains ($\overline{PAS}$).

To emulate the effect of PAS, we modified the Ropper code to extend each gadget length by one byte, decode the gadget, and check if the new gadget is semantically equivalent to the old one or not. This emulates the effect of an attacker targeting a particular address, but instead executing the one before due to the PAS security shift, δ. As shown in Table 6.1, PAS foils all the gadget-chains found by our modified Ropper. Extending the Ropper chain-creation functionality to the ARM ISA is part of our future work. Intuitively, the results would be even worse for the attacker in ARM as the state-space is more constrained due to instruction alignment requirements.

Bench. Name	$\overline{PAS}$ Chains	PAS Chains	Bench. Name	$\overline{PAS}$ Chains	PAS Chains	Bench. Name	$\overline{PAS}$ Chains	PAS Chains
perlbench	17	0	x264	23	0	lbm	23	0
gcc	23	0	deepsjeng	11	0	blender	23	0
mcf	11	0	leela	15	0	imagick	23	0
omnetpp	23	0	xz	11	0	nab	23	0
xalancbmk	15	0	namd	23	0	povray	23	0

TABLE 6.1: ROP gadget-chain reduction for SPEC2017 C/C++ benchmarks. $\overline{PAS}$ and PAS correspond to the number of valid ROP chains before and after PAS.

CONTROL-FLOW HIJACKING EVALUATION. We further evaluate security by using RIPE [193], an open source intrusion prevention benchmark suite. We port RIPE to ARM and run it on our modified Gem5, with $n = 8$ bits, as described in § 6.8. We mainly focus on return-address manipulation as a target code pointer and `ret2libc`/ROP as attack payloads. Shellcode attacks are not considered as we expect W^X.

Our ported RIPE benchmark contains 54 (relevant) attack combinations. On an unprotected Gem5 system, 50 attacks succeed and 4 attacks fail. After deploying PAS, all of the 54 attacks fail including the single-gadget `ret2libc` attacks. That is mainly due to our high number of phantoms present at runtime, $2^8 = 256$.

That said, real-world exploits typically involve payloads with several gadgets. According to Cheng *et al.* [194] the shortest gadget chain consists of thirteen gadgets. Hence, the probability for successful execution of a gadget chain is $p_{success} \leq \left(\frac{1}{256}\right)^{13} = 4.93 \times 10^{-32}$. Snow *et al.* [139] successfully exploited a vulnerability with a ROP payload consisting of only six gadgets, which would equate to a better, but still low, success probability of $p_{success} = \left(\frac{1}{256}\right)^{6} = 3.55 \times 10^{-15}$.

6.7.3 QUALITATIVE ANALYSIS

BLIND RETURN-ORIENTED PROGRAMMING. BROP attacks can remotely find ROP gadgets, in network-facing applications, without prior knowledge of the target binary [195]. The idea is to find enough gadgets to invoke the `write` system call through trial and error; then, the target binary can be copied from memory to the network to find more gadgets. As a proof of concept, the authors showed an example with 5-gadgets that invokes `write`. With PAS, the success probability of invoking `write` would be $\left(\frac{1}{256}\right)^{5} = 9.09 \times 10^{-13}$. Note that completing an end-to-end attack requires harvesting, and using, even more gadgets, after dumping the target binary, which makes the attack unfeasible on a PAS-hardened system. Additionally, BROP requires services that restart after a crash, while failed attempts will be noticeable to a system admin.

Whole-function Reuse. Unlike ROP attacks, which (re)use short instruction sequences, entire functions are invoked to manipulate the control-flow of a program. This type of attack includes counterfeit object-oriented programming (COOP) attacks, in which whole C++ functions are invoked through code pointers in read-only memory, such as `vtables` [169]. PAS relies on PtrEnc to prevent the attacker from manipulating pointers (`vptr`) that point to `vtables`—a necessary step for mounting a COOP attack.

`Ret2libc` is another example for whole function reuse attacks, in which the attacker tries to execute entire `libc` functions [196, 197].[3] With PAS, the attacker will have to guess the address of the first basic block of the function in order to lunch the attack, reducing the success probability to $\left(\frac{1}{256}\right) = 0.0039$.

Our analysis of real-world exploits shows that executing a `ret2libc` attack incurs multiple steps in order for the attacker to (i) prepare the function arguments based on the calling convention, (ii) jump to the desired function entry, (iii) silence any side-effects that occur due to executing the whole function, and (iv) reliably continue (or gracefully terminate) the victim program without noticeable crashes. Items (i) and (iii) generally requires code-reuse (ROP) gadgets, as demonstrated by the following publicly-available exploits: (a) ROP + `ret2libc`-based exploit against mcrypt [198], (b) ROP + `ret2libc`-based exploit against Nginx [199], (c) ROP + `ret2libc` + shellcode-based exploit for Apache + PHP [200] and (d) ROP + `ret2libc`-based exploit against Netperf [201]. Thus, if the ROP part of the exploit requires G gadgets, the probability for successfully exploiting the program would exponentially decrease to $p_{success} \leq \left(\frac{1}{256}\right)^G$. That is because the attacker will have to guess the domain of execution (out of $2^8 = 256$ phantoms) of every gadget.

Side-channel Attacks. PAS takes multiple steps to be resilient to side channel attacks. Firstly, PAS purposefully avoids timing variances introduced due to hardware modifications, in order to limit timing-based side channel attacks. Additionally, the attacker cannot leak the random phantom index, p, which are generated by the selector as it is unreadable from both user and kernel mode—it exists within the processor only. Similarly, the execution domain cannot be leaked to the attacker through the architectural stack, as PAS keeps it within the hardware in the SDS.

[3] In general, any function, of any other shared library, or even the main binary itself, can be used instead.

6.8 Evaluation

In this section, we devise experiments to analyze the benefits of each aspect of the PAS scheme and compare it against prior solutions.

As we focus on resource constrained devices, we use the `ARM` ISA to demonstrate PAS due to its dominance in the embedded (see § 3.2.1 for more discussion) with its 32-bit `ARMv5-8` instruction set architecture (ISA). However, the concept of PAS can be applied to any other ISA (e.g., RISC-V).

6.8.1 Experimental Setup

Core	ARMv7a OoO core at 1.8 GHz BPred: BiModeBP, 4096-entry BTB, 48-entry RAS Fetch: 3 wide, 48-entry IQ Issue: 8 wide, 60-entry ROB Writeback: 8 wide, 16-entry LQ, 16-entry SQ
L1 I-cache	32KB, 2-way, 2 cycles, 64B blocks, LRU replacement, 2 MSHRs, no prefetch
L1 D-cache	32KB, 2-way, 2 cycles, 64B blocks, LRU replacement, 16-entry write buffer, 6 MSHRs, no prefetch
L2 cache	2MB, 16-way, 15 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 16 MSHRs, stride prefetch
DRAM	LPDDR3, 1600 MHz, 1GB, 15ns CAS latency and row precharge, 42ns RAS latency

Table 6.2: PAS Simulation parameters.

We implement PAS in the out-of-order (OoO) CPU model of Gem5 [202] for the `ARM` architecture. We execute `ARM32` binaries from the SPEC CPU2017 [203] C/C++ benchmark suite on the modified simulator in syscall emulation mode with the `ex5_big` configuration (see Table 6.2), which is based on the `ARM` Cortex-A15 32-bit processor.

To compile the benchmarks, we build a complete toolchain based on a modified Clang/LLVM v7.0.0 compiler including `musl` [204], `compiler-rt`, `libunwind`, `libcxxabi`, and `libcxx`. Using a full toolchain allows us to instrument all binary code including shared libraries to remove them from the trusted code base (TCB). In order to evaluate PAS, we use our modified toolchain to generate the following variants.

Baseline. This is the case of an unmodified unprotected machine. Specifically, we compile and run the SPEC CPU2017 benchmarks using an unmodified version of the toolchain and Gem5 simulator. In all of our experiments, we use the total number of cycles (numCycles) to complete the program, as reported by Gem5, to report performance. The numCycles values of the defenses are normalized to this baseline implementation without defenses; thus, a normalized value greater than one indicates higher performance overheads.

PAS. In this scenario, we run unmodified binaries on our modified Gem5 implementation.

PAS-TRAP. In order to prepare PAS-TRAP binaries, we implement an LLVM backend pass to insert TRAPs at the beginning of BBLs. This step can also be achieved with an appropriate binary rewriter making it compatible with legacy binaries [205, 206, 207]. In our compiler pass, we modified our PAS address translation function in Gem5 to avoid executing the inserted TRAP instructions for normal program execution while resolving the $BBL_{Phantoms}$ correctly to a single BBL in the virtual address space.

PAS-PtrEncLite. To evaluate the performance of PAS with PtrEnc, we first write an LLVM IR pass to instrument the code (including shared libraries) and insert the relevant instructions as described in CCFI [168]. Specifically, we emit instructions whenever (1) a new object is created (to encrypt the contents of the vptr), (2) a virtual function call is made (to decrypt the vptr), or (3) any operation on code pointers in C programs. Then, we appropriate the encodings for ARM's ldc and stc instructions respectively, which are themselves unimplemented in Gem5, to behave as ENCP and DECP instructions. We add a dedicated functional unit in Gem5 to handle these instruction's latency in order to avoid any contention on the regular functional units. We also assume equal cycle counts of 8 for both instructions [191]. This latency is to emulate the effect of the actual encryption/decryption.

PtrEncFull. In this approach, we instrument code pointer load/store operations in addition to function entry/exit points to protect return addresses for non-leaf functions. Conceptually, this solution is similar to ARM PAC [88]. However, due to the absence of PAC support in Gem5 (and for 32-bit ARM architectures in general), we only perform behavioral simulation for comparison purposes, without keeping track of the actual pointer metadata.

Execution Path Randomization (EPR). For the sake of completeness and fair comparison, we also implement a static version where there are two copies of the code, i.e., a version without the phantom aspect of the naming scheme. In this model, we have two virtual addresses for each instruction but these addresses are physically stored in memory, essentially halving the capacity of the microarchitectural structures. [4] We create the two copies by introducing a shift of `TRAP` instruction size in one of them. At a high-level our implementation works as follows: (1) clone functions using an LLVM IR pass, (2) LLVM backend pass to insert `TRAP`s for cloned functions, (3) instruct the LLVM backend to globalize BBL labels, (4) emit a diversifier BBL for every BBL, and (5) rewrite branch instruction targets to point to the diversifier.

Of the 16 `C/C++` benchmarks, 14 compile with all different toolchain modifications. `parest` has compatibility issues with `musl` due to exception handling usages, while `povray` failed to run on Gem5. For EPR, `gcc`, `xalancbmk`, and `x264` present compilation and/or linking issues.

6.8.2 Performance

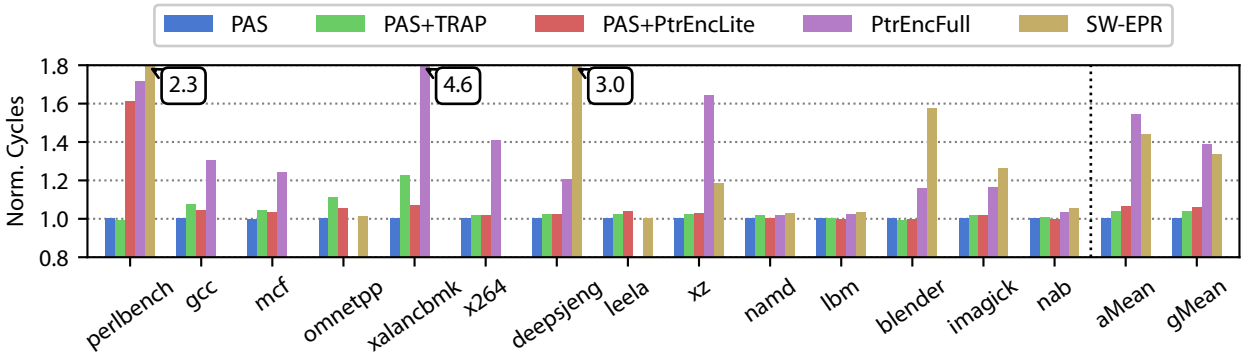


Figure 6.6: PAS performance evaluation for SPEC CPU2017

We run all benchmarks to completion with the `test` input set on our augmented Gem5. We verified the correctness of the outputs against the reference output. Figure 6.6 shows the performance overhead of the different design approaches (all normalized to Baseline). As expected, PAS has identical performance to Baseline. The overhead of PAS-`TRAP` is minimal, approximately 4%. Adding support for PtrEnc increases the performance overheads of PAS-PtrEncLite to 6%. The `perlbench` benchmark suffers from a relatively high overhead due to its extensive use of function pointers and indirect branches. On the other hand, fully

[4] This model is similar to Isomeron [147], with the modification that it is used at the BBL granularity as opposed to the original work, which uses a dynamic binary rewriting framework with function granularity.

protecting the binaries with a deterministic defense such as PtrEncFull encounters a 91% overhead on average (geometric mean of 62%). Our static implementation of software EPR introduces an arithmetic average overhead of 31% (geometric mean of 26%)[5]. In contrast to software Isomeron [147] which relies on dynamic binary instrumentation (DBI), the overheads for our implementation are primarily attributed to the indirection every BBL branch must make to the diversifier.

Benchmark Name	Call Depth	Benchmark Name	Call Depth
perlbench	24	xz	16
gcc	28	namd	12
mcf	28	parest	-
omnetpp	196	lbm	10
xalancbmk	77	blender	23
x264	15	imagick	22
deepsjeng	48	nab	16
leela	244	povray	-

Table 6.3: Maximum call depth for SPEC CPU2017 C/C++ benchmark suite.

Finally, the call depths listed in Table 6.3 show that SPEC programs do not exceed a depth of 244 (`leela`), indicating that a 256-entry hardware Secret Domain Stack is likely sufficient to handle typical program execution.

6.9 Deployment Considerations

For completeness, we outline design changes required to deploy a PAS system.

Sizing Although SDS only stores eight bits per return address in hardware, it still has a limited size that cannot be dynamically increased in contrast to the architectural stack. A statically sized SDS means that programs with deeply nested function calls may result in a SDS overflow. To handle overflows, we add two new hardware exception signals: *hardware-stack-overflow* and *hardware-stack-underflow*. The former is raised when the SDS overflows. In this case, the OS (or another trusted entity), encrypts and copies the contents of the SDS to the kernel memory. This kernel memory location will be a stack of stacks and every time a stack is full it will be appended to the previous full stack. The second exception will be raised when the SDS is empty to decrypt and page-in the last saved full-stack from kernel memory.

[5] Our EPR implementation does not instrument external libraries (only the main application code) due to compilation issues. This leads to overheads that are less than intuitively expected.

Stack Unwinding Since addresses are split across the architectural (software) stack and the SDS it is vital to keep them in sync for correct operation. Earlier, we described how normal LIFO `call`/`ret`s are handled. In some cases, however, the stack can be reset arbitrarily by `setjmp`/`longjmp` or C++ exception handling. To ensure the stack cannot be disclosed/manipulated maliciously during non-LIFO operations, we change the runtime to encrypt the `jmp_buffer` before storing it to memory. Additionally, we also store the current index of the SDS. When a `longjmp` is executed, we decrypt the contents of the `jmp_buffer` and use the decrypted SDS index to re-synchronize it with the architectural stack. The same approach can be applied to the C++ exception handling mechanism by instrumenting the appropriate APIs.

Context Switches The SDS of the current process is stored in the Process Control Block before a context switch. In terms of cost, the typical size of the SDS is 256-bytes (256 entries, each has 8-bits). Moving this number of bytes between the SDS and memory during context switch requires just a few `load` and `store` instructions, which consume a few cycles. This overhead is negligible with respect to the overhead of the rest of the context switch (which happens infrequently; every tens of milliseconds).

Dynamic Linking Dynamically-linked shared libraries are essential to modern software as they reduce program size and improve locality. Although most embedded system software (the primary target in this work) in MCUs is typically statically-linked, we note that PAS is compatible with shared libraries as it can be fully realized in hardware. Thus, PAS does not differentiate between BBLs related to the main program and the ones corresponding to shared libraries. On the other hand, dynamic linking has been a challenge for many CFI solutions, as control flow graph edges that span modules may be unavailable statically. CCFI [168] suffers from the same limitation as the dynamically shared library code needs to be instrumented before execution; otherwise, the respective pages will be vulnerable to code pointer manipulation attacks.

Debugging Support Debugging tools typically include a dump of the stack contents to help the developers understand the root cause of user programs crashes. In our secure PAS environment, the return addresses that are stored in the software stack are split to prevent the attacker from leaking information regarding the current domain of execution. To provide a consistent view during debugging, we can use a trusted system call that can be invoked only through the debugger with the capability of reconstructing the correct stack contents by reading the contents of SDS and concatenating them with the return addresses stored in the architectural stack before generating the report.

6.10 Related Work

N-Variant Execution Systems The general idea of N-Variant eXecution (NVX) systems is to run *N different* copies/variants of the same code, alongside each other, while checking their runtime behavior [166, 208]. If the variants produce a different response to a single common input (due to an internal failure or external attack payload), the checker detects such divergences in execution and raises an alert. Since 2006, many NVX systems have been proposed to achieve reliability and security goals [209, 210, 211, 212, 213, 214].

Berger and Zorn proposed a system for probabilistic memory safety that could simultaneously execute identical variants with uniquely seeded randomizing memory allocators [208]. This system only supported applications that received input through `stdin` and sent output to `stdout`. Cox *et al.* introduced N-variant systems, which can monitor a wider array of system calls (syscalls) and replicate input/output from various sources [166], effectively supporting more complex applications, such as web servers.

Specifically, Volckaert *et al.* [209] aimed at preventing control-flow hijacking attacks by replicating execution with a partitioned address space. As they synchronize all syscalls, they introduce a significant performance overhead. While follow-up proposals managed to improve the efficiency of replicated execution [209, 211, 212], such systems do not guarantee the detection of memory disclosure (e.g., MvArmor [211] cannot detect out-of-bound reads related to the stack, while ReMon [209] cannot detect memory disclosures caused by out-of-bound reads based on relative addresses).

An alternative N-variant solution, proposed by Lu *et al.* [214], dubbed BUDDY, can detect various memory disclosures by properly enforcing data diversification schemes for the variants: i.e., BUDDY maintains two running instances of the program and diversifies their target data. Memory disclosure is detected as it will result in the two instances outputting different values. Such a technique can be used to prevent the first step of various ("just-in-time") CRAs, which rely on a memory disclosure to identify (the location of) code gadgets [139]. However, BUDDY suffers from considerable runtime overhead (at least 100%) and requires program recompilation. In the same vein, Österlund *et al.* [165] recently presented kMVX, a N-variant-based defense against information leakage vulnerabilities, in kernel settings, which relies on executing multiple diversified kernel variants (simultaneously) on the same machine. kMVX incurs a 20% slowdown, on average; 50% in the worst case.

In general, NVX systems are effective at stopping attacks that rely on precise knowledge of the virtual address space layout of a target (e.g., CRAs whose payloads include absolute addresses of instruction sequences).

However, they suffer from considerable runtime performance and memory overheads, due to duplicated execution and code/data, and therefore are not suitable for resource constrained systems. PAS provides a lighter-weight alternative that ensures security and is deployable across a wide range of devices.

CRA Mitigations for Resource Constrained Devices Nyman *et al.* [215] introduced `CaRE`, an interrupt aware control-flow integrity (CFI) scheme for low-end microcontrollers that leverages TrustZone-M security extensions. `CaRE` instruments binaries in a manner which removes all function calls and indirect branches and replaces them with dispatch instructions that trap control flow to a branch monitor. Although the branch monitor eliminates control-flow attacks that utilize such branches, the performance overheads introduced range between 13% and 513%. In the case of indirect calls, `CaRE` matches the branch target against a record of valid subroutine entry points. Unlike PAS-PtrEncLite, this coarse-grained approach does not protect against whole function reuse attacks.

Live Randomization Recent work has pioneered the use of hardware moving target defenses to protect against code-reuse attacks. Gallagher *et al.* [170] proposed Morpheus, an architecture that (i) displaces code and data pointers in the address space (ii) diversifies the representation of code, code pointers, and data pointers using strong encryption, and (iii) periodically repeats the above steps using a different displacement and key.

The main conceptual difference between Morpheus and PAS is that in PAS, at any given instant there are multiple names (addresses) for an instruction while there is only one name (address) for an instruction in Morpheus. This distinction is also true of PAS and software moving target systems [216] used to protect against code reuse attacks.

In terms of security, Morpheus must keep two parameters a secret until they are changed: displacements for the code and data regions, and keys for encrypting/decrypting pointers. In the basic PAS there are no secrets, and in the enhanced PAS, there is only one secret viz. the key used for code pointer encryption. If PAS is added to Morpheus it increases the security level offered by Morpheus because the attacker has to disclose the displacement *and* break the name confusion to mount a code-reuse attack.

PAS can also provide an illusion of a faster churn rate. The churn time can be thought of as the time an attacker has to deploy a countermeasure. PAS, forces the attacker to have a counter strategy every basic block which normally completes execution in the order of nanoseconds. While Morpheus' churn rate (milliseconds

for PAS level of performance) is sufficient to protect against remote network adversaries, the (apparently) faster churn provided by PAS is meaningful in offering protection against local attackers especially with side channel capabilities, and thus is again complementary to Morpheus. The BBL-by-BBL apparent churn offered by PAS also comes at much lower energy cost compared to Morpheus as it does not require memory scanning to identify pointers. Finally, from a deployment perspective, a unique benefit of PAS is that it works for non-64-bit systems while Morpheus and software moving target systems, rely on the availability of a 64-bit address space for security.

Memory Safety Defenses Hardware primitives for memory safety, such as LowFat [217], CHERI [218], REST [219], and Califorms [220], can mitigate CRAs by detecting the initial memory safety violation. While providing higher security guarantees than PAS, the above techniques are less suitable for resource constrained systems due to their high energy overheads in addition to their intrusive changes to the entire software stack (including software, hardware, and OS). On the contrary, PAS is able to enhance security even for legacy binaries.

7 Cache Line Formats (Califorms)

Current software and hardware-supported solutions excel at providing coarse-grained, inter-object memory safety which involves detecting memory accesses beyond arrays and heap allocated regions (`malloc`'d struct and class instances). However, they are not suitable for fine-grained memory safety (i.e., intra-object memory safety—detecting overflows within objects, such as fields within a struct, or members within a class) due to the high performance overheads and/or need for making intrusive changes to the source code [221]. Moreover, existing solutions are rarely practical or deployable on 32-bit processors that dominate the CPS space. Some real-world scenarios where intra-object memory safety problems manifest are type confusion vulnerabilities (e.g., CVE–2017–5115 [222]) and uninitialized data leaks through padding bytes (e.g., CVE–2014–1444 [223]), both recognized as high-impact security classes [224, 225].

For instance, a recent work that aims to provide intra-object overflow protection functionality incurs a 2.2x performance overhead [226]. These overheads are problematic because they not only reduce the number of pre-deployment tests that can be performed, but also impede post-deployment continuous monitoring, which researchers have pointed out is necessary for detecting benign and malicious memory safety violations [227]. Thus, a low overhead memory safety solution that can enable continuous monitoring and provide complete program safety has been elusive.

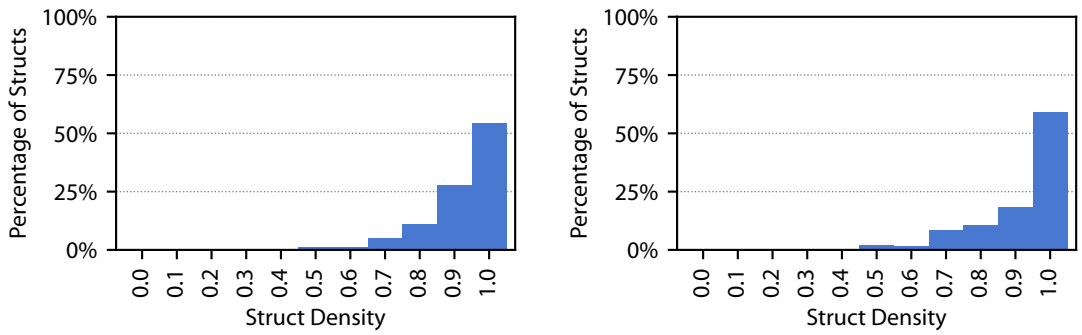
The source of overheads stem from how current designs store and use metadata necessary for enforcing memory safety. In Intel MPX [91], Hardbound [228], CHERI [229, 230], and PUMP [231], the metadata is stored for each pointer; this metadata consumes the already scarse memory on CPS devices. Additionally, since C/C++ memory accesses tend to be highly pointer based, the performance and energy overheads of accessing metadata to perform checks can be significant. The management of metadata, especially if it is stored in a disjoint manner from the pointer, can also create significant engineering complexity in terms of performance and usability. This complexity was evidenced by the fact that compilers like LLVM and GCC dropped support for Intel MPX in their mainline after an initial push to integrate it into the toolchain [91].

7.1 Overview

Califorms is a memory access control primitive that efficiently manages it's metadata by storing it inline with the program's data. By storing metadata inline, Califorms is able to overcome the drawbacks of previous techniques especially on resource constrained devices with limited memory. Caliform's approach for reducing overheads is two-fold. First, instead of checking access bounds for each pointer access, we blacklist all memory locations that should never be accessed. This reduces the additional work for memory safety such as comparing bounds. Second, we propose a novel metadata storage scheme for storing blacklisted information. We observe that by using dead memory spaces (or *dead bytes*) in the program, we can store metadata needed for memory safety for free for nearly half of the program objects making Califorms suitable for memory constrained CPSs. These dead spaces can occur for several reasons including language alignment requirements (more commonly referred to as *padding* bytes).

```
1 struct A {
2   char c;
3   /* compiler inserts padding
4   * bytes for alignment */
5   // ie. char padding_bytes[3];
6   int i;
7   char buf[64];
8   void (*fp)();
9 };
```

Listing 7.1: Example C/C++ struct highlighting padding bytes.



(a) SPEC CPU2006 C and C++.

(b) V8 JavaScript engine.

Figure 7.1: Struct density histogram of SPEC CPU2006 benchmarks and the V8 JavaScript engine.

Prevalence of Dead Bytes So how often do dead bytes occur in programs? Before we answer this question, let us concretely understand padding bytes with an example. Consider the `struct A` defined in List-

ing 7.1. Let us say the compiler inserts a three-byte padding in between `char c` and `int i` because of the C language requirement that integers should be padded to their natural size (which we assume to be four bytes here). To obtain a quantitative estimate on the amount of paddings, we developed a compiler pass to statically collect the padding size information. Figure 7.1 presents the histogram of struct densities for SPEC CPU2006 C and C++ benchmarks and the V8 JavaScript engine. Struct density is defined as the sum of the size of each field divided by the total size of the `struct` including the padding bytes (i.e., the smaller or sparse the struct density the more padding bytes the struct has). The results reveal that 45.7% and 41.0% of structs within SPEC and V8, respectively, have at least one byte of padding. This is encouraging since even without introducing additional padding bytes (no memory overhead), we can offer protection for certain compound data types restricting the remaining attack surface. When we cannot find naturally occurring dead spaces, we can manually insert them.

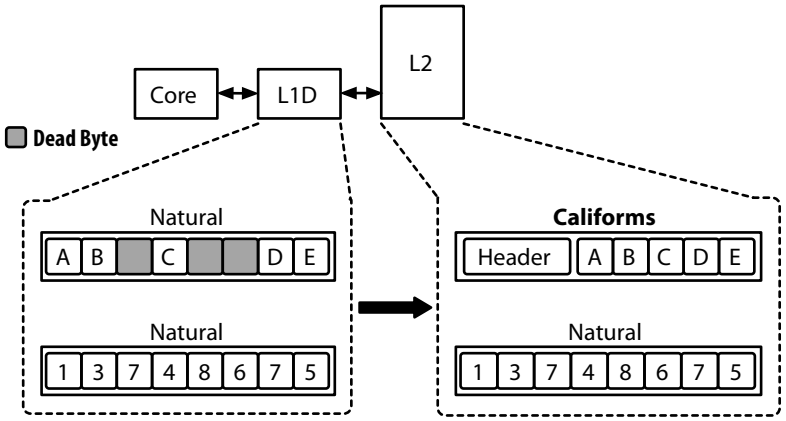
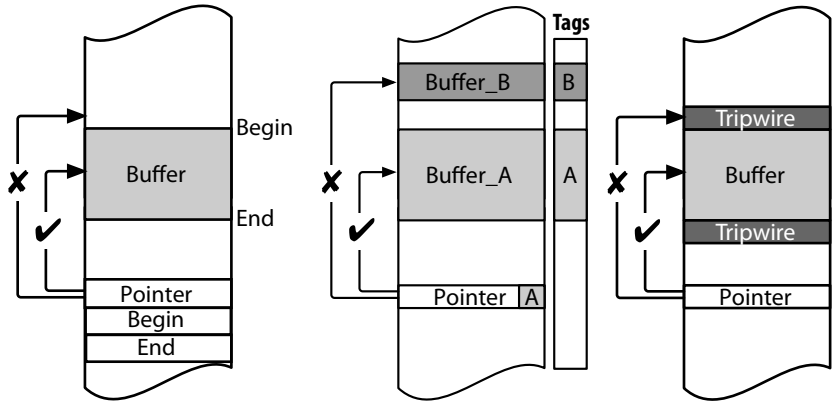


Figure 7.2: Califorms offers memory safety by detecting accesses to dead bytes in memory. Dead bytes are not stored beyond the L1 data cache and identified using a special header in the L2 cache (and beyond) resulting in very low overhead. The conversion between these formats happens when lines are filled or spilled between the L1 and L2 caches. The absence of dead bytes results in the cache lines stored in the same natural format across the memory system.

Distinguishing Dead Bytes A natural question to ask is how dead bytes are distinguished from normal bytes in memory. A straightforward scheme results in one bit of additional storage per byte to identify if a byte is a dead byte; this scheme results in a space overhead of 12.5%. We reduce this overhead to one bit per 64B cache line (0.2% overhead) without any loss of precision by only reformatting how data is stored in cache lines. Our technique, *Califorms*, uses one bit of additional storage to identify if the cache line associated with the memory contains any dead bytes. For *califormed* cache lines, i.e., lines which contain dead bytes, the actual data is stored following a "header", which indicates the location of dead bytes, as shown in Figure 7.2.



(A) Disjoint metadata whitelisting. (B) Cojoined metadata whitelisting. (C) Inlined metadata blacklisting.

FIGURE 7.3: Three main classes of hardware solutions for memory safety.

With this support, it is easy to describe how a Califorms based system for memory safety works. Dead bytes, either naturally harvested or manually inserted, are used to indicate memory regions that should never be accessed by a program (i.e., blacklisting). If an attacker accesses these regions, we detect this rogue access without any additional metadata accesses as our metadata resides inline.

7.2 Background

Hardware support for memory safety can be broadly categorized into three classes as presented in Figure 7.3.

DISJOINT METADATA WHITELISTING This class of techniques, also called base and bounds, attaches bounds metadata with every pointer, bounding the region of memory they can legitimately dereference (see Figure 7.3a). Hardbound [228], proposed in 2008, provides spatial memory safety using this mechanism. Intel MPX [91], productized in 2016, is similar and introduces an explicit architectural interface (registers and instructions) for managing bounds information. Temporal memory safety was introduced to this scheme by storing an additional "version" information along with the pointer metadata and verifying that no stale versions are ever retrieved [232, 233]. BOGO [234] adds temporal memory safety to MPX by invalidating all pointers to freed regions in MPX's lookup table. Introduced about 35 years ago in commercial chips like Intel 432 and IBM System/38, CHERI [229] revived capability based architectures with similar bounds-checking

guarantees, in addition to having other metadata fields (e.g.,permissions).[1] PUMP [231], on the other hand, is a general-purpose framework for metadata propagation, and can be used for propagating pointer bounds.

One advantage of per pointer metadata stored separately from the pointer in a shadow memory region is that it allows compatibility with codes that use legacy pointer layouts. In principle, metadata storage overhead scales according to the number of pointers, but implementations generally reserve a fixed chunk of memory for easy lookup. Owing to this disjoint nature, metadata access requires additional memory operations, which some proposals seek to minimize with caching and other optimizations. Regardless, disjoint metadata introduces atomicity concerns potentially resulting in false positives and negatives or complicating coherence designs at the least (e.g.,MPX is not thread-safe). Explicit specification of bounds per pointer also allows bounds-narrowing in principle, wherein pointer bounds can be tailored to protect individual elements in a composite memory object. However, commercial compilers do not support this feature for MPX due to the complexity of compiler analyses required. Furthermore, compatibility issues with untreated modules (e.g.,unprotected libraries) also introduces real-world deployability concerns for these techniques. For instance, MPX drops its bounds when protected pointers are modified by unprotected modules, while CHERI does not support it at all. MPX additionally makes bounds checking explicit, thus introducing a marginal computational overhead to bounds management as well.

Cojoined Metadata Whitelisting Originally introduced in the IBM System/370 mainframes, this mechanism assigns a "color" to memory chunks when they are allocated, and the same color to the pointer used to access that region. The runtime check for access validity simply consists of comparing the colors of the pointer and accessed memory (see Figure 7.3b).

This technique is currently commercially deployed by Oracle as ADI [235],[2] which uses higher order bits in pointers to store the color. In ADI, color information associated with memory is stored in dedicated per line metadata bits while in cache and in extra ECC bits while in memory [227]. The use of ECC bits creates a restriction on the number of colors, however, if the colors can fit into ECC, metadata storage does not occupy any additional memory in the program's address space.[3] Additionally, since metadata bits are acquired along with concomitant data, extra memory operations are obviated. For the same reason, it is also compatible

[1] A recent version of CHERI [230], however, manages to compress metadata to 128 bits and changes pointer layout to store it with the pointer value (i.e.,implementing base and bounds as cojoined metadata whitelisting), accordingly introducing instructions to manipulate them specifically.

[2] ARM has a similar upcoming Memory Tagging feature [89], whose implementation details are unclear, as of this work.

[3] However, when a memory is swapped color bits are copied into memory by the OS.

with unprotected modules since the checks are implicit as well. Temporal safety is achieved by assigning a different color when memory regions are reused. However, intra-object protection or bounds-narrowing is not supported as there is no means for "overlapping" colors. Furthermore, protection is also dependent on the number of metadata bits employed, since it determines the number of colors that can be assigned. So, while color reuse allows ADI to scale and limit metadata storage overhead, it can also be exploited by this vector. Another disadvantage of this technique, specifically due to inlining metadata in pointers, is that it only supports 64-bit architectures. Narrower pointers would not have enough spare bits to accommodate color information.

Inlined Metadata Blacklisting Another line of work, also referred to as tripwires, aims to detect overflows by simply blacklisting a patch of memory on either side of a buffer, and flagging accesses to this patch (see Figure 7.3c). This is very similar to contemporary canary design [175], but there are a few critical differences. First, canaries only detect overwrites, not overreads. Second, hardware tripwires trigger instantaneously, whereas canaries need to be periodically checked for integrity, providing a period of attack to time of use window. Finally, unlike hardware tripwires, canary values can be leaked or tampered, and thus mimicked.

SafeMem [236] implements tripwires by repurposing ECC bits in memory to mark memory regions invalid, thus trading off reliability for security. On processors supporting speculative execution, however, it might be possible to speculatively fetch blacklisted lines into the cache without triggering a faulty memory exception. Unless these lines are flushed immediately after, SafeMem's blacklisting feature can be trivially bypassed. Alternatively, REST [219] achieves the same by storing a predetermined large random number, in the form of an 8–64B token, in the memory to be blacklisted. Violations are detected by comparing cache lines with the token when they are fetched. REST provides temporal memory safety by quarantining freed memory, and not reusing them for subsequent allocations. Compatibility with unprotected modules is easily achieved as well, since tokens are part of the program's address space and all access are implicitly checked. However, intra-object safety is not supported by REST owing to fragmentation overhead such heavy usage of tokens would entail.

Since Califorms operates on the principle of detecting memory accesses to dead bytes, which are in turn stored along with program data, it belongs to the inlined metadata class of defenses. However, it is different from other works in the class in one key aspect—granularity. While both REST and SafeMem naturally black-

list at the cache line granularity, Califorms can do so at the byte granularity. It is this property that enables us to provide intra-object safety with negligible performance and memory overheads, unlike previous work in the area. For inter-object spatial safety and temporal safety, we employ the same design principles as REST. Hence, our safety guarantees are a *strict superset* of those provided by previous schemes in this class (spatial memory safety by blacklisting and temporal memory safety by quarantining).

7.3 Threat Model

We assume a threat model comparable to that used in contemporary related works [219, 229, 230]. We assume the victim program to have one or more vulnerabilities that an attacker can exploit to gain arbitrary read and write capabilities in the memory; our goal is to prevent both spatial and temporal memory violations. Furthermore, we assume that the adversary has access to the source code of the program, therefore she is able to glean all source-level information. However, she does not have access to the host binary (e.g., server-side applications). Finally, we assume that all hardware is trusted—it does not contain and/or is not subject to bugs arising from exploits such as physical or glitching attacks. Due to its recent rise in relevance however, we maintain side-channel attacks in our design of Califorms within the purview of our threats. Specifically, we accommodate attack vectors seeking to leak the location and value of security bytes.

7.4 Framework

The Califorms framework consists of multiple components we discuss in the following sections:

- Architecture Support. A new instruction called `BLOC`, mnemonic for Blacklist LOCations, that blacklists memory locations at byte granularity and raises a privileged exception upon misuse of blacklisted locations (§ 7.4.1).
- Microarchitecture Design. New cache line formats, or Califorms, that enable low cost access to the metadata—we propose different Caliform for L1 cache vs. L2 cache and beyond (§ 7.4.2).
- Software Design. Compiler, memory allocator and operating system extensions which insert the security bytes at compile time and manages them via the `BLOC` instruction at runtime (§ 7.4.3).

At compile time, each compound data type (e.g., `struct` or `class`) is examined and security bytes are added according to a user defined insertion policy viz. opportunistic, full, or intelligent, by a source-to-source translation pass. At execution time when compound data type instances are dynamically created in the heap, we use a new version of `malloc` that issues `BLOC` instructions to arrange the security bytes after the space is allocated. When the `BLOC` instruction is executed, the cache line format is transformed at the L1 cache controller (assuming a cache miss) and is inserted into the L1 data cache. Upon an L1 eviction, the L1 cache controller transforms the cache line to meet the Caliform of the L2 cache.

While we add additional metadata storage to the caches, we refrain from doing so for main memory and persistent storage to keep the changes local within the CPU core. When a califormed cache line is evicted from the last-level cache to main memory, we keep the cache line califormed and store the additional one metadata bit into spare ECC bits similar to Oracle's ADI [227, 235].[4] When a page is swapped out from main memory, the page fault handler stores the metadata for all the cache lines within the page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. Therefore, our design keeps the cache line format califormed throughout the memory hierarchy. A califormed cache line is un-califormed only when the corresponding bytes cross the boundary where the califormed data cannot be understood by the other end, such as writing to I/O (e.g.,pipe, filesystem or network socket). Finally, when an object is freed, the freed bytes are filled with security bytes and quarantined to offer temporal memory safety. At runtime, when a rogue load or store accesses a security byte the hardware returns a privileged, precise security exception to the next privilege level which can take any appropriate action including terminating the program.

7.4.1 Architecture Support

BLOC Instruction

		R2, R3		
		X, $\overline{\text{Allow}}$	$\overline{\text{Set}}$, Allow	Set, Allow
Initial	Regular Byte	Regular Byte	Exception	Security Byte
	Security Byte	Security Byte	Regular Byte	Exception

Table 7.1: BLOC instruction K-map. X represents "Don't Care".

[4] ADI stores four bits of metadata per cache line for allocation granularity enforcement while Caliform stores one bit for sub-allocation granularity enforcement.

The format of the instruction is "BLOC R1, R2, R3". The value in register R1 points to the starting (64B cache line aligned) address in the virtual address space, denoting the start of the 64B chunk which fits in a single cache line. Table 7.1 represents a K-map for the BLOC instruction. The value in register R2 indicates the attributes of said region represented in a bit vector format (1 to set and 0 to unset the security byte). The value in register R3 is a mask to the corresponding 64B region, where 1 allows and 0 disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line. We throw a privileged exception when the BLOC instruction tries to set a security byte to an existing security byte, or unset a security byte from a normal byte.

The BLOC instruction is treated similarly to a store instruction in the processor pipeline since it modifies the architectural state of data bytes in a cache line. It first fetches the corresponding cache line into the L1 data cache upon an L1 miss (assuming a write allocate cache policy). Next, it manipulates the bits in the metadata storage to appropriately set or unset the security bytes.

Privileged Exceptions

When the hardware detects an access violation (i.e.,access to a security byte), it throws a privileged exception once the instruction becomes non-speculative. There are some library functions which violate the aforementioned operations on security bytes such as memcpy so we need a way to suppress the exceptions. In order to whitelist such functions, we manipulate the exception mask registers and let the exception handler decide whether to suppress the exception or not. Although privileged exception handling is more expensive than handling user-level exceptions (because it requires a context switch to the kernel), we stick with the former to limit the attack surface. We rely on the fact that the exception itself is a rare event and would have negligible effect on performance.

7.4.2 Microarchitecture Design

The microarchitectural support for our technique aims to keep the common case fast: L1 cache uses the straightforward scheme of having one bit of additional storage per byte. All califormed cache lines are converted to the straightforward scheme at the L1 data cache controller so that typical loads and stores which hit in the L1 cache do not have to perform address calculations to figure out the location of original data (which is required for Califorms in the L2 cache and beyond). This design decision guarantees that the common case latencies will not be affected due to security functionality. Beyond the L1, the data is stored in

the optimized califormed format, i.e.,one bit of additional storage for the entire cache line. The transformation happens when the data is filled in or spilled from the L1 data cache (between the L1 and L2), and adds minimal latency to the L1 miss latency.

L1 Cache: Bit Vector Approach

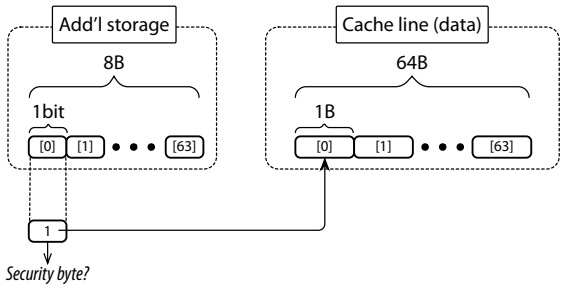


Figure 7.4: Califorms-bitvector: L1 Califorms implementation using a bit vector that indicates whether each byte is a security byte. HW overhead of 8B per 64B cache line.

To satisfy the L1 design goal we consider a naive (but low latency) approach which uses a bit vector to identify which bytes are security bytes in a cache line. Each bit of the bit vector corresponds to each byte of the cache line and represents its state (normal byte or security byte). Figure 7.4 presents a schematic view of the *Califorms-bitvector* implementation. The bit vector requires a 64-bit (8B) bit vector per 64B cache line which adds 12.5% storage overhead for the L1 data cache (comparable to ECC overhead for reliability). If a load accesses a security byte (which is determined by reading the bit vector) an exception is recorded to be processed when the load is ready to be committed. Meanwhile, the load returns a pre-determined value for the security byte (in our design the value zero which is the value that the memory region is initialized to upon deallocation). Returning this fixed value is meant to be a countermeasure against speculative side-channel attacks that seek to identify security byte locations (discussed in greater detail in § 7.5). On store accesses to security bytes, we report an exception when the store commits.

L2 Cache and Beyond: Sentinel Approach

For L2 and beyond, we take a different approach that allows us to recognize whether each byte is a security byte with fewer bits, as using the L1 metadata format throughout the system will increase the cache area overhead by 12.5%, which may not be acceptable. We propose *Califorms-sentinel*, which has a 1-bit or 0.2% metadata overhead per 64B cache line. For main memory, we store the additional bit per cache line size in the

DRAM ECC spare bits, thus completely removing any cycle time impact on DRAM access or modifications to the DIMM architecture.

The key insight that enables significant memory savings is the following: *the number of bits required to address all the bytes in a cache line, which is six bits for a 64 byte cache line, is less than a single byte.* For example, let us assume that there is (at least) one security byte in a 64B cache line. Considering a byte granular protection, there are at most 63 unique values (bytes) that non-security bytes can have. Therefore, we are guaranteed to find a six bit pattern that is not present in any of the normal bytes', for instance, the least significant six bits. We use this pattern as a sentinel value to represent the security bytes in the cache line. Now if we store this six bit (sentinel value) as additional metadata, the storage overhead will be seven bits (six bits plus one bit to specify if the cache line is califormed) per cache line. We further propose a new cache line format which stores the sentinel value within a security byte to reduce the metadata overhead down to one bit per cache line.

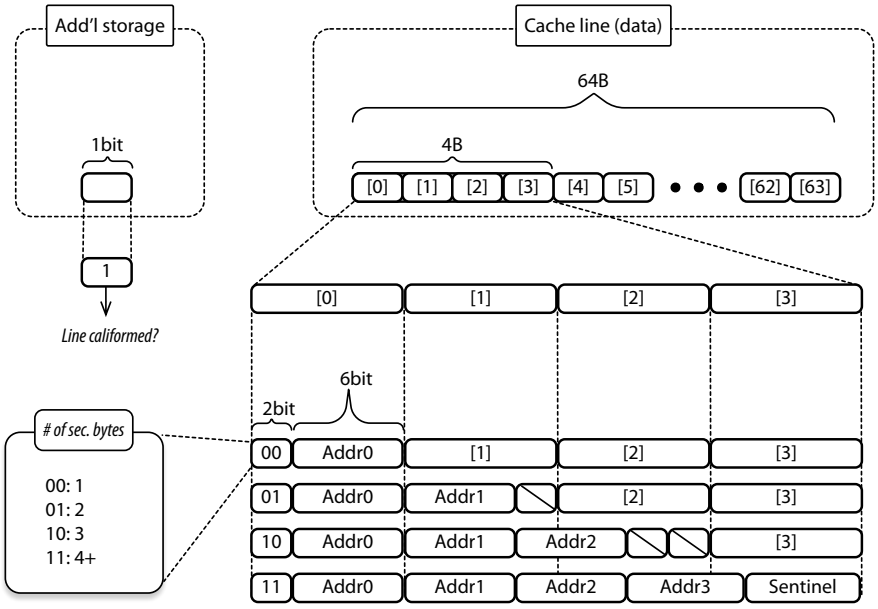


FIGURE 7.5: Califorms-sentinel that stores a bit vector in security byte locations. HW overhead of 1-bit per 64B cache line.

As presented in Figure 7.5, Califorms-sentinel stores the metadata into the first four bytes (at most) of the 64B cache line. Two bits of the first (0th) byte are used to specify the number of security bytes within the cache line: `00`, `01`, `10` and `11` represent one, two, three, and four or more security bytes, respectively. The sentinel is used only when we have more than four security bytes. If there is only one security byte in the cache line, we use the remaining six bits of the 0th byte to specify the location of the security byte, and the original value

of the 0th byte is stored in the security byte. Similarly, when there are two or three security bytes in the cache line, we use the bits of the second and third bytes to locate them. The key observation is that, we gain two bits per security byte since we only need six bits to specify a location in the cache line. Therefore, when we have four security bytes we can locate four addresses and have six bits remaining in the first four bytes. This remaining six bits can be used to store a sentinel value, which allows us to have any number of additional security bytes.

Although the sentinel value depends on the actual values within the 64B cache line, it works naturally with a write-allocate L1 cache (which is the most commonly used cache allocation policy in modern microprocessors). The cache line format is transformed upon L1 cache eviction and insertion (Califorms-bitvector to/from Califorms-sentinel), while the sentinel value only needs to be found upon L1 cache eviction (L1 miss). Also, it is important to note that Califorms-sentinel supports critical-word first delivery since the security byte locations can be quickly retrieved by scanning only the first 4B of the first 16B flit.

L1 to/from L2 Califorms Conversion

Figure 7.6 and Figure 7.7 describe the high-level process used for converting from L1 to L2 Califorms and vice-versa.

The spill to L2 (i.e., L1 to L2 conversion) works as follows. First, we determine whether the evicted line contained security bytes, if so, enabling the Califorms bit. For Califormed lines, we scan the least significant 6-bits of every byte to determine an appropriate sentinel value. We then proceed to find the locations of the security bytes, consisting of taking the entire cache line and searching for the index of the first zero. We perform this search in parallel (four times for four security bytes). Finally, we store the data of the first four bytes in the locations found above.

The fill to L1 (i.e., L2 to L1 conversion) is simpler than the spill. The Califorms bit of the L2 inserted line is used to control the value of the L1 cache (Califorms-bitvector) metadata. The first two bits of the L2 inserted line are used as inputs to decide on the metadata bits of the first four bytes. Only if those two bits are `11` is the sentinel value read from the fourth byte and fed, with the least signficant 6-bits of each byte.

7.4.3 Software Design

We describe the memory allocator, compiler and the operating system changes to support Califorms in the following section.

```
 1: Read the Caliform metadata for the evicted line and OR them
 2: if result is 0 then
 3:    Evict the line as is and set Caliform bit to zero
 4: else
 5:    Set Caliform bit to one
 6:    if num security bytes (N) < 4 then
 7:       Get locations of first N security bytes
 8:       Store data of first N bytes in locations obtained in 7
 9:       Fill the first N bytes based on Figure 7.5
10:    else
11:       Get locations of first four security bytes
12:       Scan least 6-bit of every byte to determine sentinel
13:       Store data of first four bytes in locations obtained in 11
14:       Fill the first four bytes based on Figure 7.5
15:       Use the sentinel to mark the remaining security bytes
16:    end
17: end
```

FIGURE 7.6: Califorms conversion from the L1 cache (Califorms-bitvector) to L2 cache (Califorms-sentinel).

```
 1: Read the Caliform bit for the inserted line
 2: if result is 0 then
 3:    Set the Caliform metadata bit vector to [0]
 4: else
 5:    Check the least significant 2-bit of byte 0
 6:    Set the metadata of byte[Addr[0-3]] to one based on 5
 7:    Set the metadata of byte[Addr[byte==sentinel]] to one
 8:    Set the data of byte[0-3] to byte[Addr[0-3]]
 9:    Set the new locations of byte[Addr[0-3]] to zero
10: end
```

FIGURE 7.7: Califorms conversion from the L2 cache (Califorms-sentinel) to L1 cache (Califorms-bitvector).

Dynamic Memory Management

We can consider two approaches to applying security bytes: (i) *Dirty-before-use.* Unallocated memory has no security bytes. We set security bytes upon allocation and unset them upon deallocation; or (ii) *Clean-before-use.* Unallocated memory remains filled with security bytes all the time. We clear the security bytes (in legitimate data locations) upon allocation and set them upon deallocation.

Ensuring temporal memory safety in the heap remains a non-trivial problem [237]. We therefore choose to follow a *clean-before-use* approach in the heap, so that deallocated memory regions remain protected by security bytes.[5] In order to provide temporal memory safety (to mitigate use-after-free exploits), we do not reallocate recently freed regions until the heap is sufficiently consumed (quarantining). Additionally, both ends of the heap allocated regions are protected by security bytes in order to provide inter-object memory safety. Compared to the heap, the security benefits of clean-before-use are limited for the stack since temporal

[5] It is natural to use a variant of `BLOC` instruction which bypasses (does not store into) the L1 data cache, just like the non-temporal (or streaming) load/store instructions (e.g.,`MOVNTI`, `MOVNTQ`, etc) when deallocating a memory region; deallocated region is not meant to be used by the program and thus polluting the L1 data cache with those memory is harmful and should be avoided. However, we do not evaluate the use of such instructions.

attacks on the stack (e.g.,use-after-return attacks) are much rarer. Hence, we apply the *dirty-before-use* scheme on the stack.

Compiler Support

Our compiler-based instrumentation infers where to place security bytes within target objects, based on their type layout information. The compiler pass supports three insertion policies: the first *opportunistic* policy supports security bytes insertion into existing padding bytes within the objects, and the other two support modifying object layouts to introduce randomly sized security byte spans that follow the *full* or *intelligent* strategies described in § 7.5.1. The first policy aims at retaining interoperability with external code modules (e.g.,shared libraries) by avoiding type layout modification. Where this is not a concern, the latter two policies help offer stronger security coverage—exhibiting a tradeoff between security and performance.

Operating System Support

We need the following support in the operating system:

- Privileged Exceptions. As the Califorms exception is privileged, the operating system needs to properly handle it as with other privileged exceptions (e.g.,page faults). We also assume the faulting address is passed in an existing register so that it can be used for reporting/investigation purposes. Additionally, for the sake of usability and backwards compatibility, we have to accommodate copying operations similar in nature to `memcpy`. For example, a simple `struct` to `struct` assignment could trigger this behavior, thus leading to a potential breakdown of software with Califorms support. Hence, in order to maintain usability, we allow whitelisting functionality to suppress the exceptions. This can either be done with a privileged store (requiring a syscall) or an unprivileged store. Both options represent different design points in the performance-security tradeoff spectrum.

- Page Swaps. As we have discussed in § 7.1, data with security bytes is stored in main memory in a califormed format. When a page with califormed data is swapped out from main memory, the page fault handler needs to store the metadata for the entire page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. The kernel has enough address space in practice (kernel's virtual address space is 128TB for a 64-bit Linux with 48-bit virtual address space) to

store the metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

7.5 Security Discussion

7.5.1 Security Byte Policies

```
struct A {
  char c;
  int i;
  char buf[64];
  void (*fp)();
}
```

(A) Original.

```
struct A_opportunistic {
  char c;
  /* compiler inserts padding
   * bytes for alignment */
  char padding_bytes[3];
  int i;
  char buf[64];
  void (*fp)();
}
```

(B) Opportunistic.

```
struct A_full {
  /* we protect every field with
   * random security bytes */
  char security_bytes[2];
  char c;
  char security_bytes[1];
  int i;
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[1];
}
```

(C) Full.

```
struct A_intelligent {
  char c;
  int i;
  /* we protect boundaries
   * of arrays and pointers with
   * random security bytes */
  char security_bytes[3];
  char buf[64];
  char security_bytes[2];
  void (*fp)();
  char security_bytes[3];
}
```

(D) Intelligent.

Figure 7.8: (a) Original source code and examples of three security bytes harvesting strategies: (b) *opportunistic* uses the existing padding bytes as security bytes, (c) *full* protect every field within the struct with security bytes, and (d) *intelligent* surrounds arrays and pointers with security bytes.

To offer protection for all defined compound data types, we can insert random sized padding bytes, also referred to as security bytes, between every field of a `struct` or member of a `class` as in Figure 7.8c (full strategy). Random sized security bytes are chosen to provide a probabilistic defense as fixed sized security bytes can be jumped over by an attacker once she identifies the actual size (and the exact memory layout). By carefully choosing the minimum and maximum of random sizes, we can keep the average size of security bytes small (few bytes). Intuitively, the higher the unpredictability (or randomness) within the memory layout, the higher the security level we can offer.

While the full strategy provides the widest coverage, not all of the security bytes provide the same security utility. For example, basic data types such as `char` and `int`, cannot be overflowed past their bounds. The idea behind the intelligent insertion strategy is to prioritize insertion of security bytes into security-critical locations as shown in Figure 7.8d. We choose data types which are most prone to abuse by an attacker via overflow type accesses: (1) arrays and (2) data and function pointers. In Figure 7.8d, the array `buf[64]` and the function pointer `fp` are protected with random sized security bytes. While it is possible to utilize padding

bytes present between other data types without incurring memory overheads, doing so would come at an additional performance overhead.

In comparison to opportunistic harvesting, the more secure strategies (full and intelligent) come at an additional performance overhead. We analyze the performance trend in order to decide how many security bytes can be reasonably inserted. For this purpose, we developed an LLVM pass which pads every field of a `struct` and member of a `class` with fixed size paddings. We measure the performance of SPEC CPU2006 benchmarks by varying the padding size from one byte to seven bytes (since eight bytes is the finest granularity that state-of-the-art technique can offer [219]). The detailed evaluation environment and methodology are described later in § 7.6.

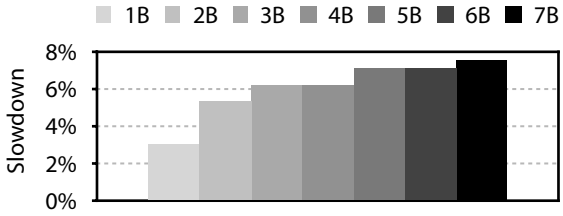


Figure 7.9: Average performance overhead with additional paddings (one byte to seven bytes) inserted for every field within structs (and classes) of SPEC CPU2006 C and C++ benchmarks.

Figure 7.9 demonstrates the average slowdown when inserting additional bytes for harvesting. As expected, we can see the performance overheads grow as we increase the padding size, mainly due to ineffective cache usage. On average the slowdown is 3.0% for one byte and 7.6% for seven bytes of padding. The figure presents the ideal (lower bound) performance overhead when fully inserting security bytes into compound data types; the hardware and software modifications we introduce add additional overheads on top of these numbers. We strive to provide a mechanism that allows the user to tune the security level at the cost of performance and thus explore several security byte insertion strategies to reduce the performance overhead.

7.5.2 Hardware Attacks and Mitigations

Metadata Tampering Attacks A key feature of Califorms as a metadata-based safety mechanism is the absence of programmer visible metadata in the general case (apart from a metadata bit in the page information maintained by higher privilege software). Beyond the implications for its storage overhead, this also means that our technique is immune to attacks that explicitly aim to leak or tamper the metadata to bypass the

respective defense. This, in turn, implies a smaller attack surface so far as software maintenance of metadata is concerned.

Bit-granularity Attacks Califorms's capability of fine-grained memory protection is the key enabler for intra-object overflow detection. However, our byte granular mechanism is not enough for protecting bit-fields without turning them into `char` bytes functionally. This should not be a major detraction since security bytes can still be added around composites of bit-fields.

Heterogeneous Architectural Attacks Califorms' hardware modifications affect the memory hierarchy. Hence, its protection is lost whenever one of its layers is bypassed (e.g.,heterogeneous architectures or DMA is used). Mitigating this requires that these mechanisms always respect the security byte semantics by propagating them along the respective memory structures and detecting accesses to them. If the algorithm used for califorming is used by accelerators then attacks through heterogeneous components can also be averted.

Side-Channel Attacks Our design takes multiple steps to be resilient to side channel attacks. Firstly, we purposefully avoid timing variances introduced due to our hardware modifications in order to avoid timing based side channel attacks. Additionally, to avoid speculative execution side channels ala Spectre [238], our design returns zero on a load to a security byte, thus preventing speculative disclosure of metadata. We augment this further by requiring that deallocated objects (heap or stack) be zeroed out in software[239]. This is to avoid the following attack scenario: consider a case if the attacker somehow knows that the padding locations should contain a non-zero value (for instance, because s/he knows the object allocated at the same location prior to the current object had non-zero values). However, while speculatively disclosing memory contents of the object, s/he discovers that the padding location contains a zero instead. As such, s/he can infer that the padding there contains a security byte. If deallocations were accompanied with zeroing, however, this assumption does not hold.

7.5.3 Software Attacks and Mitigations

Coverage-Based Attacks For califorming the padding bytes (in an object), we need to know the precise type information of the allocated object. This is not always possible in C-style programs where `void*` allocations may be used. In these cases, the compiler may not be able to infer the correct type, in which case

intra-object support may be skipped for such allocations. Similarly, our metadata insertion policies (viz., intelligent and full) require changes to the type layouts. This means that interactions with external modules that have not been compiled with Califorms support may need (de)serialization to remain compatible. For an attacker, such points in execution may appear lucrative because of inserted security bytes getting stripped away in those short periods. We note however that the opportunistic policy can still remain in place to offer some protection. On the other hand, for those interactions that remain oblivious to type layout modifications (e.g.,passing a pointer to an object that shall remain opaque within the external module), our hardware-based implicit checks have the benefit of persistent tampering protection, even across binary module boundaries.

Whitelisting Attacks Our concession of allowing whitelisting of certain functions was necessary to make Califorms more usable in common environments without requiring significant source modifications. However, this also creates a vulnerability window wherein an adversary can piggy back on these functions in the source to bypass our protection. To confine this vector, we keep the number of whitelisted functions as minimal as possible.

Derandomization Attacks Since Califorms can be bypassed if an attacker can guess a security bytes location, it is crucial that security bytes be placed unpredictably. For the attacker to carry out a guessing attack, s/he first needs to obtain the virtual memory address of the object they want to corrupt, and then overwrite a certain number of bytes within that object. To know the address of the object of interest, s/he typically has to scan the process' memory: the probability of scanning without touching any of the security bytes is $(1 - P/N)^O$ where O is number of allocated objects, N is the size of each object, and P is number of security bytes within that object. With 10% padding ($P/N = 0.1$), when O reaches 250, the attack success goes to 10^{-20}. If the attacker can somehow reduce O to 1, which represents the ideal case for the attacker, the probability of guessing the element of interest is $1/7^n$ (since we insert 1–7 wide security bytes), compounding as the number of paddings to be guessed ($= n$) increases.

The randomness is, however, introduced statically akin to the `randstruct` plugin introduced in recent Linux kernels which randomizes structure layout of those which are specified (it does not offer detection of rogue accesses unlike Califorms do)[240, 241]. The static nature of the technique may make it prone to brute force attacks like BROP[195] which repeatedly crashes the program until the correct configuration is guessed. This could be prevented by having multiple versions of the same binary with different padding sizes

or simply by better logging, when possible. Another mitigating factor is that BROP attacks require specific type of program semantics, namely, automatic restart-after-crash with the same memory layout. Applications with these semantics can be modified to spawn with a different padding layout and yet satisfy application level requirements.

7.5.4 Pairing with PAS

Califorms and PAS can be used together to further strengthen security while remaining performant. Both techniques provide additional depth in the unlikely event an attacker randomly bypasses either protection. PAS provides the largest benefit when Califorms uses the opportunistic security byte policy. PAS can provide protection when calling uninstrumented code (e.g., shared libraries that are not instrumented with `BLOC`) as it requires no software modifications. It is important to note that the benefits are not one sided; Califorms can also be used to strengthen PAS. For instance, security bytes can be used to extend the benefits of `TRAP` instructions and other fine-grained tripwire mechanisms that provide more precise failure modes.

7.6 Evaluation

Here, we evaluate the overheads incurred by the software based changes required to enable inter-/intra-object and temporal memory safety with Califorms: the effect of underutilized memory structures (e.g.,caches) due to additional security bytes, the additional work necessary to issue `BLOC` instructions (and the overhead of executing the instructions themselves), and the quarantining to support temporal memory safety.

Experimental Setup We run the experiments on an Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We omit `dealII` and `omnetpp` due to library compatibility issues in our evaluation environment, and `gcc` since it fails when executed with the memory allocator with inter-object spatial and temporal memory safety support. The remaining 16 SPEC CPU2006 C/C++ benchmarks are compiled with our modified Clang version 6.0.0 with "`-O3 -fno-strict-aliasing`" flags. We use the `ref` inputs and run to completion. We run each benchmark-input pair five times and use the shortest execution time as its performance. For benchmarks with multiple `ref` inputs, the sum of the execution

time of all the inputs are used as their execution times. We use the arithmetic mean to represent the average slowdown.[6]

We estimate the performance impact of executing a BLOC instruction by emulating it with a dummy store instruction that writes some value to the corresponding cache line's padding byte. Since a single BLOC instruction is able to caliform the entire cache line, issuing one dummy store instruction per to-be-califormed cache line suffices. In order to issue the dummy stores, we implement a LLVM pass to instrument the code to hook into memory allocations and deallocations. We then retrieve the type information to locate the padding bytes, calculate the number of dummy stores and the address they access, and finally emit them. Therefore, all the software overheads we need to pay to enable Califorms are accounted for in our evaluation.

For the random sized security bytes, we evaluate three variants: we fix the minimum size to one byte while varying the maximum size to three, five and seven bytes (i.e.,on average the amount of security bytes inserted are two, three and four bytes, respectively). In addition, in order to account for the randomness introduced by the compiler, we generate three different versions of binaries for the same setup (e.g.,three versions of `astar` with random sized paddings of minimum one byte and maximum three bytes). The error bars in the figure represent the minimum and the maximum execution times among 15 executions (three binaries × five runs) and the average of the execution times is represented as the bar.

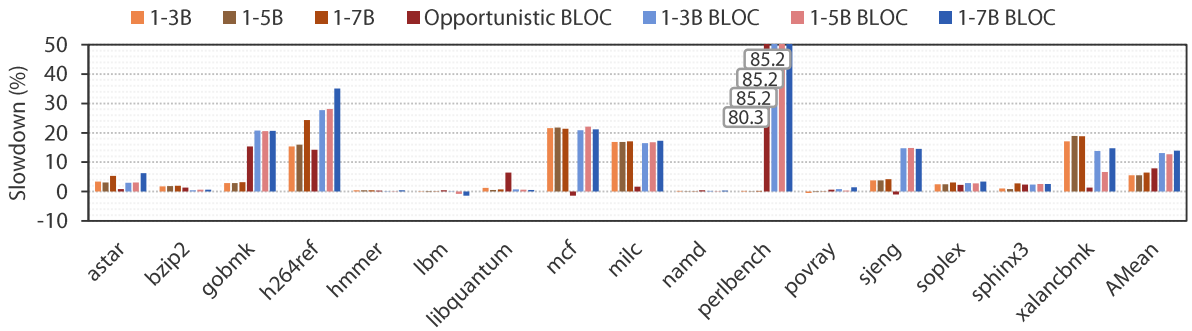


FIGURE 7.10: Slowdown of the opportunistic policy, and full insertion policy with random sized security bytes (with and without BLOC instructions). The average slowdowns of opportunistic and full insertion policies are 6.2% and 14.2%, respectively.

PERFORMANCE OF THE OPPORTUNISTIC AND FULL INSERTION POLICIES WITH BLOC INSTRUCTIONS Figure 7.10 presents the slowdown incurred by three set of strategies: full insertion policy (with random sized

[6]The use of arithmetic mean of the speedup (execution time of the original system divided by that of the system with additional latency) means that we are interested in a condition where the workloads are not fixed and all types of workloads are equally probable on the target system [242, 243].

security bytes) *without* BLOC instructions, the opportunistic policy *with* BLOC instructions, and the full insertion policy *with* BLOC instructions. Since the first strategy does not execute BLOC instructions it does not offer any security coverage, but is shown as a reference to showcase the performance breakdown of the third strategy (cache underutilization vs. executing BLOC instructions).

First, we focus on the three variants of the first strategy, which are shown in the three left most bars. We can see that different sizes of (random sized) security bytes does not make a large difference in terms of performance. The average slowdown of the three variants are 5.5%, 5.6% and 6.5%, respectively. This can be backed up by our results shown in Figure 7.9, where the average slowdowns of additional padding of two, three and four bytes ranges from 5.4% to 6.2%. Therefore in order to achieve higher security coverage without losing performance, using a random sized bytes of, minimum of one byte and maximum of seven bytes, is promising. When we focus on individual benchmarks, we can see that a few benchmarks including `h264ref`, `mcf`, `milc` and `omnetpp` incur noticeable slowdowns (ranging from 15.4% to 24.3%).

Next, we examine the opportunistic policy *with* BLOC instructions, which is shown in the middle (fourth) bar. Since this strategy does not add any additional security bytes, the overheads are purely due to the work required to setup and execute BLOC instructions. The average slowdown of this policy is 7.9%. There are benchmarks which encounter a slowdown of more than 10%, namely `gobmk`, `h264ref` and `perlbench`. The overheads are due to frequent allocations and deallocations made during program execution, where we have to calculate and execute BLOC instructions upon every event (since every compound data type requires security bytes management). For instance `perlbench` is notorious for being malloc-intensive, and reported as such elsewhere [244].

Lastly the third policy, the full insertion policy *with* BLOC instructions, offers the highest security coverage in Califorms based system with the highest average slowdown of 14.0% (with the random sized security bytes of maximum seven bytes). Nearly half (seven out of 16) the benchmarks encounter a slowdown of more than 10%, which might not be suitable for performance-critical environments, and thus the user might want to consider the use of the following intelligent insertion policy.

Performance of the Intelligent Insertion Policy with BLOC Instructions Figure 7.11 shows the slowdowns of the intelligent insertion policy with random sized security bytes (*with* and *without* BLOC instructions, in the same spirit as Figure 7.10). First we focus on the strategy without executing BLOC instructions (the three bars on the left). The performance trend is similar such that the three variants with different

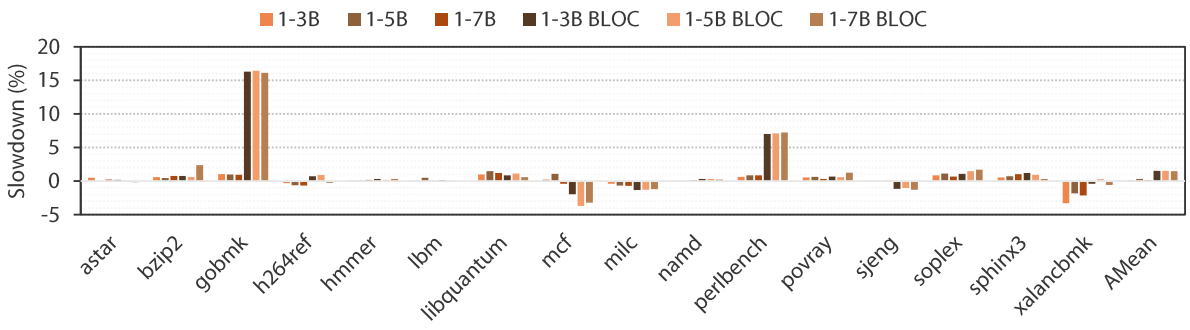


FIGURE 7.11: Slowdown of the intelligent insertion policy with random sized security bytes (with and without BLOC instructions). The average slowdown is 2.0%

random sizes have little performance difference, where the average slowdown is 0.2% with the random sized security bytes of maximum seven bytes. We can see that none of the programs incurs a slowdown of greater than 5%. Finally with BLOC instructions (three bars on the right), `gobmk` and `perlbench` have slowdowns of greater than 5% (16.1% for `gobmk` and 7.2% for `perlbench`). The average slowdown is 1.5%, where considering its security coverage and performance overheads the intelligent policy might be the most practical option for many environments.

7.7 Related Work

Tables 7.2, 7.3, and 7.4 summarize the security, performance, and implementation characteristics of the hardware based memory safety techniques discussed in § 7.2, respectively. Califorms has the advantage of requiring simpler hardware modifications and being faster than disjoint metadata based whitelisting systems. The hardware savings mainly stem from the fact that our metadata resides with program data; it does not require explicit propagation while additionally obviating all lookup logic. This significantly reduces our design's implementation costs. Califorms also has lower performance and energy overheads since it neither requires multiple memory accesses, nor does it incur any significant checking costs. However, Califorms can be bypassed if accesses to security bytes can be avoided. This safety-vs.-complexity tradeoff is critical to deployability and we argue that our design point is more practical. This is because designers have to contend with integrating these features to already complicated processor designs, without introducing additional bugs while also keeping the functionality of legacy software intact. This is a hard balance to strike [91].

Proposal	Protection Granularity	Intra-Object	Binary Composability	Temporal Safety
Hardbound [228]	Byte	✓*	✗	✗
Watchdog [232]	Byte	✓*	✗	✓
WatchdogLite [233]	Byte	✓*	✗	✓
Intel MPX [91]	Byte	✓*	✗‡	✗
BOGO [234]	Byte	✓*	✗‡	✓
PUMP [231]	Word	✗	✓	✓
CHERI [229]	Byte	✗†	✗	✗
CHERI concentrate [230]	Byte	✗†	✗	✗
SPARC ADI [235]	Cache line	✗	✓	✓§
SafeMem [236]	Cache line	✗	✓	✗
REST [219]	8--64B	✗	✓	✓¶
Califorms	Byte	✓	✓	✓¶

*Achieved with bounds narrowing.
†Although the hardware supports bounds narrowing, CHERI foregoes it since doing so compromises capability logic [245].
‡Execution compatible, but protection dropped when external modules modify pointer.
§Limited to 13 tags.
¶Allocator should randomize allocation predictability.

Table 7.2: Califorms security comparison against prior hardware techniques.

On the other hand, ideal cojoined metadata mechanisms would have comparable slowdowns and similar compiler requirements. However, practical implementations like ADI exhibits some crucial differences from the ideal.

- It is limited to 64-bit architectures, which excludes a large portion of embedded and IoT processors that operate on 32-bit or narrower platforms.
- It has finite number of colors since available tag bits are limited—ADI supports 13 colors with 4 tag bits. This is important because reusing colors proportionally reduces the safety guarantees of these systems in the event of a collision.
- It operates at the coarse granularity of cache line width, and hence, is not practically applicable for intra-object safety.

On the contrary, Califorms is agnostic of architecture width and is better suited for deployment over a more diverse device environment as is common in the CPS domain. In terms of safety, collision is not an issue for

Proposal	Metadata Overhead	Memory Overhead	Performance Overhead	Main Operations
Hardbound [228]	0--2 words per ptr, 4b per word	$\approx$ # of ptrs and prog memory footprint	$\approx$ # of ptr derefs	1--2 mem ref for bounds (may be cached), check μops.
Watchdog [232]	4 words per ptr	$\approx$ # of ptrs and allocations	$\approx$ # of ptr derefs	1--3 mem ref for bounds (may be cached), check μops.
WatchdogLite [233]	4 words per ptr	$\approx$ # of ptrs and allocations	$\approx$ # of ptr ops	1--3 mem ref for bounds (may be cached), check & propagate insns.
Intel MPX [91]	2 words per ptr	$\approx$ # of ptrs	$\approx$ # of ptr derefs	2+ mem ref for bounds (may be cached), check & propagate insns.
BOGO [234]	2 words per ptr	$\approx$ # of ptrs	$\approx$ # of ptr derefs	MPX ops + ptr miss exception handling, page permission mods.
PUMP [231]	64b per cache line	$\approx$ Prog memory footprint	$\approx$ # of ptr ops	1 mem ref for tags, may be cached, fetch and chk rules; propagate tags.
CHERI [229]	256b per ptr	$\approx$ # of ptrs and physical mem	$\approx$ # of ptr ops	1+ mem ref for capability (may be cached), capability management insns.
CHERI concentrate [230]	Ptr size is 2x	$\approx$ # of ptrs	$\approx$ # of ptr ops	Wide ptr load (may be cached), capability management insns.
SPARC ADI [235]	4b per cache line	$\approx$ Prog memory footprint	$\approx$ # of tag (un)set ops	(Un)set tag.
SafeMem [236]	2x blacklisted memory	$\approx$ Blacklisted memory	$\approx$ # of ECC (un)set ops	Syscall to scramble ECC, copy data content.
REST [219]	8--64B token	$\approx$ Blacklisted memory	$\approx$ # of arm/disarm insns	Execute arm/disarm insns.
Califorms	Byte granular security byte	$\approx$ Blacklisted memory	$\approx$ # of `BLOC` insns.	Execute `BLOC` insns.

TABLE 7.3: Califorms performance comparison against previous hardware techniques.

our design. Hence, unlike cojoined metadata systems, our security does not scale inversely with the number of allocations in the program. Finally, our fine-grained protection makes us suitable for intra-object memory safety which is a non-trivial threat in modern security [225].

Proposal	Core	Caches/TLB	Memory	Software
Hardbound [228]	μop injection & logic for ptr meta, extend reg file and data path to propagate ptr meta	Tag cache and its TLB	N/A	Compiler & allocator annotates ptr meta
Watchdog [232]	μop injection & logic for ptr meta, extend reg file and data path to propagate ptr meta	Ptr lock cache	N/A	Compiler & allocator annotates ptr meta
WatchdogLite [233]	N/A	N/A	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
Intel MPX [91]	Unknown (closed platform [246], design likely similar to Hardbound)			Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
BOGO [234]	Unknown (closed platform[246], design likely similar to Hardbound)			MPX mods + kernel mods for bounds page right management
PUMP [231]	Extend all data units by tag width, modify pipeline stages for tag checks, new miss handler	Rule cache	N/A	Compiler & allocator (un)sets memory, tag ptrs
CHERI [229]	Capability reg file, coprocessor integrated with pipeline	Capability caches	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
CHERI concentrate [230]	Modify pipeline to integrate ptr checks	N/A	N/A	Compiler & allocator annotates ptrs, compiler inserts meta propagation and check insns
SPARC ADI [235]	Unknown (closed platform)			Compiler & allocator (un)sets memory, tag ptrs
SafeMem [236]	N/A	N/A	Repurposes ECC bits	Original data copied to distinguish from hardware faults
REST [219]	N/A	1--8b per L1D line, 1 comparator	N/A	Compiler & allocator (un)sets tags, allocator randomizes allocation order/placement
Califorms	N/A	8b per L1D line, 1b per L2/L3 line	Use unused/spare ECC bit	Compiler & allocator mods to (un)set tags, compiler inserts intra-object spacing

Table 7.4: Califorms implementation complexity comparison against previous hardware techniques.

Part IV

Conclusion

8 Conclusion

Resource constraints and economic pressures in the CPS domain are at ends with the goal of security. However, as cyber-physical systems further integrate into the many facets of daily life, finding ways of ensuring CPSs remain secure and free of harm is a pressing concern. There is a need for low overhead security solutions that are deployable to harden resource constrained CPSs and increase adoption. To this end, this dissertation puts forth the thesis that by leveraging fundamental physical properties, and extending age-old computing abstractions, defenses can be designed to address the challenges of the CPS domain.

In this dissertation, we propose three techniques with various levels of security coverage and deployability. Most importantly, these techniques are tailor-made to account for the resource limitations of CPSs. The first, YOLO (discussed in Chapter 5), leverages the physical property of inertia to ensure the stability of a system while resets are performed, effectively mitigating against software memory attacks. In contrast to prior work on CPSs, which focus mostly on sensors, our defense leverages both cyber and physical properties for a bespoke defense against software threats. We show that new, simple to implement, low-resource defenses are possible if we leverage the unique physical properties of CPSs. In addition to the theoretical framework presented in § 5.4, YOLO was experimentally evaluated on two real-world CPSs further validating its applicability.

The other two approaches, PAS and Califorms revisit age-old computing abstractions, re-evaluating their meaning through the lens of CPS security. PAS (discussed in Chapter 6) introduces a new architectural addressing mechanism on top of the virtual address space by assigning multiple names (or phantoms) to every instruction. PAS's new addressing scheme is used to provide execution path randomization to thwart code-reuse attacks with close to 0% performance overheads. Moreover, the microarchitectural changes are simple and lightweight; PAS does not depend on "free" bits or the vastness of the 64-bit address space to work, making it suitable for 32-bit microcontrollers. Thus, PAS provides a cheaply deployable hardware technique that strengthens control-flow protection uniformly across resource constrained devices. Califorms (discussed in

Chapter 7) proposes a new cache based compression scheme that is leveraged to provide memory safety. The key observation behind Califorms is that a blacklisted memory region need not store useful data separately in most cases, since we can utilize byte-granular, existing or added, space present between object elements to store metadata. Califorms's in-place compact data structure avoids consuming additional memory by reusing dead memory for blacklisting. The in-place metadata scheme additionally avoids overheads associated with extraneously fetching memory from multiple locations making it very performant. Califorms' low memory overhead effectively brings fine grained memory safety to a whole new class of resource constrained devices.

The state of CPSs has reached an important inflection point: an increased awareness of security has led to a search for likely threats and solutions. Our experience in developing YOLO, PAS, and Califorms, has highlighted the importance of CPSs and their properties (as discussed in Chapter 2) in order to ensure security. As CPSs continue to grow more complex, their physical reactiveness and observability will play fundamental roles in making secure systems possible.

While memory safety and other cyber layer threats will continue to persist, solutions like YOLO, PAS, and Califorms reduce their impact and significantly limit the attack surface. As both researchers and adversaries continue to refine the effectiveness of cyber-physical layer attacks, as a community we should remain attentive to their developments (we discuss the latest work in Chapter 4). In the future, we should dedicate additional resources to tackling this class of threats, in particular those focusing on sensors. As we've discussed, sensors serve as the ground truth for a CPS's state. Thus, securing sensors can provide greater assurances over their measurements. Additionally, the increased reliability of sensors may also inform the development of new cyber layer defenses that are more efficient and robust. Finally, while increasing the security of the cyber and cyber-physical layers may not be sufficient to address cross layer threat vectors (as discussed in Chapter 4) it nonetheless provides useful building blocks to achieving full system security.

References

[1] E. A. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors*, vol. 15:no. 3, pp. 4837–4869, 2015. DOI: 10.3390/s150304837. [Online]. Available: https://doi.org/10.3390/s150304837.

[2] M. Broy and A. Schmidt, "Challenges in engineering cyber-physical systems," *IEEE Computer*, vol. 47:no. 2, pp. 70–72, 2014. DOI: 10.1109/MC.2014.30. [Online]. Available: https://doi.org/10.1109/MC.2014.30.

[3] H. A. Müller, "The rise of intelligent cyber-physical systems," *IEEE Computer*, vol. 50:no. 12, pp. 7–9, 2017. DOI: 10.1109/MC.2017.4451221. [Online]. Available: https://doi.org/10.1109/MC.2017.4451221.

[4] R. Minerva, A. Biru, and D. Rotondi, "Towards a definition of the internet of things (iot)," Tech. Rep., 2015. [Online]. Available: https://iot.ieee.org/definition.html.

[5] C. Greer, M. Burns, D. Wollman, and E. Griffor, "Cyber-physical systems and internet of things," Tech. Rep., 2019. DOI: 10.6028/nist.sp.1900-202. [Online]. Available: https://doi.org/10.6028/nist.sp.1900-202.

[6] M. Wolf and D. N. Serpanos, "Safety and security in cyber-physical systems and internet-of-things systems," *Proceedings of the IEEE*, vol. 106:no. 1, pp. 9–20, 2018. DOI: 10.1109/JPROC.2017.2781198. [Online]. Available: https://doi.org/10.1109/JPROC.2017.2781198.

[7] H. Mantel, "On the composition of secure systems," in *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*, IEEE Computer Society, 2002, pp. 88–101. DOI: 10.

1109/SECPRI.2002.1004364. [Online]. Available: https://doi.org/10.1109/SECPRI.2002.1004364.

[8] J. M. Wing, "A call to action: Look beyond the horizon," *IEEE Security & Privacy*, vol. 1:no. 6, pp. 62–67, 2003. DOI: 10.1109/MSECP.2003.1253571. [Online]. Available: https://doi.org/10.1109/MSECP.2003.1253571.

[9] A. Datta, J. Franklin, D. Garg, L. Jia, and D. K. Kaynar, "On adversary models and compositional security," *IEEE Security & Privacy*, vol. 9:no. 3, pp. 26–32, 2011. DOI: 10.1109/MSP.2010.203. [Online]. Available: https://doi.org/10.1109/MSP.2010.203.

[10] W. Chou, L. Chen, J. Shao, A. Chen, R. Chung, and L. Zhou, "Semiconductors - the next wave: Opportunities and winning strategies for semiconductor companies," *Deloitte.*, 2019. [Online]. Available: https://www2.deloitte.com/tw/en/pages/technology-media-and-telecommunications/articles/semiconductor-next-wave.html.

[11] D. McCandless, *Codebases: Millions of lines of code*, 2015. [Online]. Available: http://bit.ly/KIB_linescode.

[12] S. McConnell, *Software quality at top speed*, 2019. [Online]. Available: https://stevemcconnell.com/articles/software-quality-at-top-speed/.

[13] A. A. Cárdenas, S. Amin, and S. Sastry, "Research challenges for the security of control systems," in *3rd USENIX Workshop on Hot Topics in Security, HotSec'08, San Jose, CA, USA, July 29, 2008, Proceedings*, N. Provos, Ed., USENIX Association, 2008. [Online]. Available: http://www.usenix.org/events/hotsec08/tech/full_papers/cardenas/cardenas.pdf.

[14] S. Halder, A. Ghosal, and M. Conti, "Secure OTA software updates in connected vehicles: A survey," *CoRR*, vol. abs/1904.00685, 2019. arXiv: 1904.00685. [Online]. Available: http://arxiv.org/abs/1904.00685.

[15] S. Evanczuk, *Eetimes 2019 embedded markets study*, 2019. [Online]. Available: https://www.embedded.com/2019-embedded-markets-study-reflects-emerging-technologies-continued-c-c-dominance/.

[16] E. IoT, *Iot developer survey results*, 2019. [Online]. Available: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf.

[17] P. Koopman, "Embedded system security," *IEEE Computer*, vol. 37:no. 7, pp. 95–97, 2004. DOI: 10.1109/MC.2004.52. [Online]. Available: https://doi.org/10.1109/MC.2004.52.

[18] ——, *Embedded system engineering economics*, 2015. [Online]. Available: http://users.ece.cmu.edu/~koopman/ece649/lectures/12_product_economics.pdf.

[19] M. Miller, *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*, 2019. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[20] G. P. Zero, *0day in the wild*, https://googleprojectzero.blogspot.com/p/0day.html, [Online; accessed 15-Feb-2020], 2019.

[21] Y. Park, Y. Son, H. Shin, D. Kim, and Y. Kim, "This ain't your dose: Sensor spoofing attack on medical infusion pump," in *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*, N. Silvanovich and P. Traynor, Eds., USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/park.

[22] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds., USENIX Association, 2015, pp. 881–896. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/son.

[23] Y. Tu, S. Rampazzi, B. Hao, A. Rodriguez, K. Fu, and X. Hei, "Trick or heat?: Manipulating critical temperature-based control systems using rectification attacks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., ACM, 2019, pp. 2301–2315. DOI: 10.1145/3319535.3354195. [Online]. Available: https://doi.org/10.1145/3319535.3354195.

[24] M. Guri, B. Zadov, and Y. Elovici, "Led-it-go: Leaking (A lot of) data from air-gapped computers via the (small) hard drive LED," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, M. Polychronakis and M. Meier, Eds., ser. Lecture Notes in Computer Science, vol. 10327, Springer, 2017, pp. 161–184. DOI: 10.1007/978-3-319-60876-1_8. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_8.

[25] E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," *IEEE Security & Privacy*, vol. 16:no. 1, pp. 54–62, 2018. DOI: 10.1109/MSP.2018.1331033. [Online]. Available: https://doi.org/10.1109/MSP.2018.1331033.

[26] Y. Shoukry, P. D. Martin, P. Tabuada, and M. B. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, G. Bertoni and J. Coron, Eds., ser. Lecture Notes in Computer Science, vol. 8086, Springer, 2013, pp. 55–72. DOI: 10.1007/978-3-642-40349-1_4. [Online]. Available: https://doi.org/10.1007/978-3-642-40349-1_4.

[27] D. F. Kune, J. D. Backes, S. S. Clark, D. B. Kramer, M. R. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost talk: Mitigating EMI signal injection attacks against analog sensors," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, IEEE Computer Society, 2013, pp. 145–159. DOI: 10.1109/SP.2013.20. [Online]. Available: https://doi.org/10.1109/SP.2013.20.

[28] C. Bolton, S. Rampazzi, C. Li, A. Kwong, W. Xu, and K. Fu, "Blue note: How intentional acoustic interference damages availability and integrity in hard disk drives and operating systems," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, IEEE Computer Society, 2018, pp. 1048–1062. DOI: 10.1109/SP.2018.00050. [Online]. Available: https://doi.org/10.1109/SP.2018.00050.

[29] S. Wood, G. Forsyth, and R. Mangtani, *The economics of the security of consumer-grade iot products and services*, 2019. [Online]. Available: https://plumconsulting.co.uk/the-economics-of-the-security-of-consumer-grade-iot-products-and-services/.

[30] S. D. Johnson, J. M. Blythe, M. Manning, and G. T. W. Wong, "The impact of iot security labelling on consumer product choice and willingness to pay," *PLOS ONE*, vol. 15:no. 1, pp. 1–21, 2020. DOI: 10.1371/journal.pone.0227800. [Online]. Available: https://doi.org/10.1371/journal.pone.0227800.

[31] P. Rodríguez-Dapena, "Software safety certification: A multidomain problem," *IEEE Software*, vol. 16:no. 4, pp. 31–38, 1999. DOI: 10.1109/52.776946. [Online]. Available: https://doi.org/10.1109/52.776946.

[32] G. Baldini, A. F. Skarmeta, E. Fourneret, R. Neisse, B. Legeard, and F. L. Gall, "Security certification and labelling in internet of things," in *3rd IEEE World Forum on Internet of Things, WF-IoT 2016, Reston, VA, USA, December 12-14, 2016*, IEEE Computer Society, 2016, pp. 627–632. DOI: 10.1109/WF-IoT.2016.7845514. [Online]. Available: https://doi.org/10.1109/WF-IoT.2016.7845514.

[33] M. Meriac and J. Yiu, "High-end security features for low-end microcontrollers," in *International Workshop on MILS: Architecture and Assurance for Secure Systems, MILS 2017, Nürnberg, Germany, March 14, 2017*, S. Tverdyshev, Ed., Zenodo, 2017. DOI: 10.5281/zenodo.571158. [Online]. Available: https://doi.org/10.5281/zenodo.571158.

[34] ARM, *Arm mbed uvisor*. [Online]. Available: https://www.mbed.com/en/technologies/security/uvisor/.

[35] F. Lambert, *Https://electrek.co/2016/02/26/tesla-vertically-integrated/*, 2016.

[36] A. Devices, *Analog devices advisory to ics alert-17-073-01*, 2017. [Online]. Available: https://www.analog.com/media/en/Other/Support/product-security-response/ADI_Response-ICS_Alert-17-073-01.pdf.

[37] I. Insights, *The mcclean report*, 2018. [Online]. Available: https://distributor.electronicspecifier.com/ic-insights-sees-soaring-microcontroller-sales/.

[38] C. Yan, H. Shin, C. Bolton, W. Xu, Y. Kim, and K. Fu, "Sok: A minimalist approach to formalizing analog sensor security," in *2020 IEEE Symposium on Security and Privacy, SP 2020*, IEEE Computer Society, 2020.

[39] I. Giechaskiel and K. B. Rasmussen, "Sok: Taxonomy and challenges of out-of-band signal injection attacks and defenses," *CoRR*, vol. abs/1901.06935, 2019. arXiv: 1901.06935. [Online]. Available: http://arxiv.org/abs/1901.06935.

[40] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during program execution," 1977.

[41] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Software Eng.*, vol. 11:no. 12, pp. 1491–1501, 1985. DOI: 10.1109/TSE.1985.231893. [Online]. Available: https://doi.org/10.1109/TSE.1985.231893.

[42] L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, Boston, 2001, ISBN: 9781580531375.

[43] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48:no. 1, 16:1–16:26, 2015. DOI: 10.1145/2807593. [Online]. Available: https://doi.org/10.1145/2807593.

[44] C.-G. Kang, "Origin of stability analysis: Ön governorsb̸j.c. maxwell [historical perspectives]," *IEEE Control Systems Magazine*, vol. 36:no. 5, pp. 77–88, 2016, ISSN: 1066-033X. DOI: 10.1109/mcs.2016.2584358.

[45] N. Wiener, *Cybernetics or control and communication in the animal and the machine*. M.I.T. Press, 2014.

[46] V. Askue, "Fly-by-wire," *Air medical journal*, vol. 22:no. 6, pp. 4–5, 2003.

[47] Bosch, *50 years of bosch gasoline injection jetronic*, 2020.

[48] C. Ten, C. Liu, and G. Manimaran, "Vulnerability assessment of cybersecurity for scada systems," *IEEE Transactions on Power Systems*, vol. 23:no. 4, pp. 1836–1846, 2008.

[49] F. Pasqualetti, F. Dörfler, and F. Bullo, "Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design," in *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference, CDC-ECC 2011, Orlando, FL, USA, December 12-15, 2011*, IEEE, 2011, pp. 2195–2201. DOI: 10.1109/CDC.2011.6160641. [Online]. Available: https://doi.org/10.1109/CDC.2011.6160641.

[50] J. P. Monteuuis, A. Boudguiga, J. Zhang, H. Labiod, A. Servel, and P. Urien, "SARA: security automotive risk analysis method," in *Proceedings of the 4th ACM Workshop on Cyber-Physical System Security, CPSS@AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, D. Gollmann and J. Zhou, Eds., ACM, 2018, pp. 3–14. DOI: 10.1145/3198458.3198465. [Online]. Available: https://doi.org/10.1145/3198458.3198465.

[51] M. Burmester, E. Magkos, and V. Chrissikopoulos, "Modeling security in cyberphysical systems," *International Journal of Critical Infrastructure Protection*, vol. 5:no. 3, pp. 118 –126, 2012, ISSN: 1874-5482. DOI: https://doi.org/10.1016/j.ijcip.2012.08.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874548212000443.

[52] R. Lanotte, M. Merro, A. Munteanu, and L. Viganò, "A formal approach to physics-based attacks in cyber-physical systems," *ACM Trans. Priv. Secur.*, vol. 23:no. 1, 3:1–3:41, 2020. DOI: 10.1145/3373270. [Online]. Available: https://doi.org/10.1145/3373270.

[53] N. Shevchenko, B. R. Frye, and C. Woody, "Threat modeling for cyber-physical system-of-systems: Methods evaluation," Carnegie Mellon University Software Engineering Institute, Tech. Rep., 2018.

[54] E. R. Griffor, C. Greer, D. A. Wollman, and M. J. Burns, "Framework for cyber-physical systems: Volume 1, overview," Tech. Rep., 2017. DOI: 10.6028/nist.sp.1500-201. [Online]. Available: https://doi.org/10.6028/nist.sp.1500-201.

[55] S. R. Chhetri, J. Wan, and M. A. Al Faruque, "Cross-domain security of cyber-physical systems," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 200–205.

[56] N. H. C. Guzman, M. Wied, I. Kozine, and M. A. Lundteigen, "Conceptualizing the key features of cyber-physical systems in a multi-layered representation for safety and security analysis," *Syst. Eng.*, vol. 23:no. 2, pp. 189–210, 2020. DOI: 10.1002/sys.21509. [Online]. Available: https://doi.org/10.1002/sys.21509.

[57] R. Pallas-Areny and J. G. Webster, *Sensors and signal conditioning*. John Wiley & Sons, 2012.

[58] Statista, *Worldwide operational stock of industrial robots from 2009 to 2022*, 2019. [Online]. Available: http://www.statista.com/statistics/947017.

[59] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, IEEE Computer Society, 2017, pp. 268–286. DOI: 10.1109/SP.2017.20. [Online]. Available: https://doi.org/10.1109/SP.2017.20.

[60] R. Bishop, *The Mechatronics Handbook, Second Edition - 2 Volume Set*, ser. Mechatronics Handbook 2e. Taylor & Francis, 2007, ISBN: 9780849392573. [Online]. Available: https://books.google.com/books?id=j-ZKAQAACAAJ.

[61] G. Geography, *Https://gisgeography.com/trilateration-triangulation-gps/*, 2020.

[62] K. J. resetAström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[63] C. Bormann, M. Ersue, and A. Keränen, *Terminology for Constrained-Node Networks*, RFC 7228, 2014. DOI: 10.17487/RFC7228. [Online]. Available: https://rfc-editor.org/rfc/rfc7228.txt.

[64] P. Clarke, *Mcu market turns to 32-bits and arm* -, 2013. [Online]. Available: https://www.eetimes.com/mcu-market-turns-to-32-bits-and-arm/.

[65] I. Insights, *The mcclean report*, 2020. [Online]. Available: https://www.icinsights.com/services/mcclean-report/report-contents/.

[66] E. IoT, *Iot developer survey results*, 2018. [Online]. Available: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2018.pdf.

[67] Statista, *Server unit shipments by technology worldwide from 2016 to 2022 (in millions)*, 2019. [Online]. Available: http://www.statista.com/statistics/934508.

[68] ——, *Personal computer (pc) shipments (desktop and portable/notebook) worldwide from 2009 to 2021 (in million units)*, 2020. [Online]. Available: http://www.statista.com/statistics/269049.

[69] ——, *Microcontroller unit (mcu) shipments worldwide from 2015 to 2023 (in billions)*, 2019. [Online]. Available: http://www.statista.com/statistics/935382.

[70] ——, *Microcontroller unit (mcu) market revenues from 2005 to 2020, by microcontroller type*, 2020. [Online]. Available: https://www.statista.com/statistics/553426.

[71] J. Huang, *Https://www.nxp.com/docs/en/supporting-information/bl-micro-nxp-microcontroller-overview-james-huang.pdf*, 2017.

[72] Wikipedia contributors, *Digi-key — Wikipedia, the free encyclopedia*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Digi-Key.

[73] J. Bungo, *The arm architecture (with focus on cortex-m3*, 2015. [Online]. Available: https://www.slideshare.net/ahireprashant/arm-cortexm3-byjoebungoarm?next_slideshow=1.

[74] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *19th Euromicro Conference on Real-Time Systems, ECRTS'07, 4-6 July 2007, Pisa, Italy, Proceedings*, IEEE Computer Society, 2007, pp. 169–178. DOI: 10.1109/ECRTS.2007.25. [Online]. Available: https://doi.org/10.1109/ECRTS.2007.25.

[75] C. Meenderinck, A. M. Molnos, and K. Goossens, "Composable virtual memory for an embedded soc," in *15th Euromicro Conference on Digital System Design, DSD 2012, Cesme, Izmir, Turkey, September 5-8, 2012*, IEEE Computer Society, 2012, pp. 766–773. DOI: 10.1109/DSD.2012.32. [Online]. Available: https://doi.org/10.1109/DSD.2012.32.

[76] W. Puffitsch and M. Schoeberl, "Time-predictable virtual memory," in *19th IEEE International Symposium on Real-Time Distributed Computing, ISORC 2016, York, United Kingdom, May 17-20, 2016*, IEEE Computer Society, 2016, pp. 158–165. DOI: 10.1109/ISORC.2016.30. [Online]. Available: https://doi.org/10.1109/ISORC.2016.30.

[77] NXP, *Nxp automotive safety mcus and mpus*, 2020. [Online]. Available: https://www.nxp.com/docs/en/product-selector-guide/BRAUTOPRDCTMAP.pdf.

[78] *Armv8-m memory protection unit*, version Version 2.1, ARM, 2019. [Online]. Available: https://static.docs.arm.com/100699/0201/armv8m_memory_protection_unit_100699_0201_en.pdf.

[79] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, V. Atluri, C. A. Meadows, and A. Juels, Eds., ACM, 2005, pp. 340–353. DOI: 10.1145/1102120.1102165. [Online]. Available: https://doi.org/10.1145/1102120.1102165.

[80] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August*

14-15, 2017, W. Enck and C. Mulliner, Eds., USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier.

[81] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Comput. Surv.*, vol. 52:no. 1, 18:1–18:36, 2019. DOI: 10.1145/3297714. [Online]. Available: https://doi.org/10.1145/3297714.

[82] J. Huang and J. Liaw, *Optimize uclinux for arm cortex-m4*, 2015. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems.

[83] A. Madhavapeddy and D. J. Scott, "Unikernels: The rise of the virtual library operating system," *Commun. ACM*, vol. 57:no. 1, pp. 61–69, 2014. DOI: 10.1145/2541883.2541895. [Online]. Available: https://doi.org/10.1145/2541883.2541895.

[84] *Github*. [Online]. Available: https://github.com.

[85] Wikipedia contributors, *Comparison of real-time operating systems — Wikipedia, the free encyclopedia*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems.

[86] S. Smalley and J. Carter, *Security in zephyr and fuchsia*, 2018. [Online]. Available: https://www.linux.com/training-tutorials/redefining-security-technology-zephyr-and-fuchsia/.

[87] A. Boie, *Retrofitting zephyr memory protection*, 2018. [Online]. Available: https://elciotna18.sched.com/event/DYLv/retrofitting-memory-protection-in-the-zephyr-os-andrew-boie-intel.

[88] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[89] ARM, *Memory tagging extension: Enhancing memory safety through architecture*, https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety, 2019.

[90] Intel, *Intel control-flow enforcement technology preview*, 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[91] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel MPX system stack," *POMACS*, vol. 2:no. 2, 28:1–28:30, 2018. DOI: 10.1145/3224423. [Online]. Available: https://doi.org/10.1145/3224423.

[92] J. Ganz and S. Peisert, "ASLR: how robust is the randomness?" In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, IEEE Computer Society, 2017, pp. 34–41. DOI: 10.1109/SecDev.2017.19. [Online]. Available: https://doi.org/10.1109/SecDev.2017.19.

[93] W. Bolton, *Instrumentation and Control Systems*. Newnes, 2015.

[94] H. Kopetz, A. Bondavalli, F. Brancati, B. Frömel, O. Höftberger, and S. M. Iacob, "Emergence in cyber-physical systems-of-systems (cpsoss)," in *Cyber-Physical Systems of Systems - Foundations - A Conceptual Model and Some Derivations: The AMADEOS Legacy*, ser. Lecture Notes in Computer Science, A. Bondavalli, S. Bouchenak, and H. Kopetz, Eds., vol. 10099, Springer, 2016, pp. 73–96. DOI: 10.1007/978-3-319-47590-5_3. [Online]. Available: https://doi.org/10.1007/978-3-319-47590-5_3.

[95] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the nsf workshop on formal methods for security," USA, Tech. Rep., 2016.

[96] U. I. C. E. R. Team, "Recommended practice: Improving industrial control systems cybersecurity with defense-in-depth strategies," *Department of Homeland Security, Washington, DC, USA, www. ics-*

cert. us-cert. gov/sites/default/files/recommended _practices/NCCIC_ICS-CERT_Defense_in_Depth_-2016_S508C. pdf, 2016.

[97] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44:no. 4, pp. 35–41, 2001. DOI: 10.1145/367211.367250. [Online]. Available: https://doi.org/10.1145/367211.367250.

[98] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Hybrid systems modeling challenges caused by cyber-physical systems," 2013.

[99] S. Bliudze, S. Furic, J. Sifakis, and A. Viel, "Rigorous design of cyber-physical systems," *Software & Systems Modeling*, vol. 18:no. 3, pp. 1613–1636, 2017. DOI: 10.1007/s10270-017-0642-5. [Online]. Available: https://doi.org/10.1007/s10270-017-0642-5.

[100] T. A. Henzinger, "The theory of hybrid automata," in *Verification of Digital and Hybrid Systems*, M. K. Inan and R. P. Kurshan, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 265–292, ISBN: 978-3-642-59615-5. DOI: 10.1007/978-3-642-59615-5_13. [Online]. Available: https://doi.org/10.1007/978-3-642-59615-5_13.

[101] H. Shin, Y. Son, Y. Park, Y. Kwon, and Y. Kim, "Sampling race: Bypassing timing-based analog active sensor spoofing detection on analog-digital systems," in *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*, N. Silvanovich and P. Traynor, Eds., USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/shin.

[102] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling uavs with sensor input spoofing attacks," in *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*, N. Silvanovich and P. Traynor, Eds., USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/davidson.

[103] K. Eykholt, I. Evtimov, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song, "Robust physical-world attacks on machine learning models," *CoRR*, vol. abs/1707.08945, 2017. arXiv: 1707.08945. [Online]. Available: http://arxiv.org/abs/1707.08945.

[104] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "Dolphinattack: Inaudible voice commands," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 103–117. DOI: 10.1145/3133956.3134052. [Online]. Available: https://doi.org/10.1145/3133956.3134052.

[105] J. Selvaraj, G. Y. Dayanikli, N. P. Gaunkar, D. Ware, R. M. Gerdes, and M. Mina, "Electromagnetic induction attacks against embedded systems," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, J. Kim, G. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, Eds., ACM, 2018, pp. 499–510. DOI: 10.1145/3196494.3196556. [Online]. Available: https://doi.org/10.1145/3196494.3196556.

[106] T. Hardin, R. Scott, P. Proctor, J. D. Hester, J. Sorber, and D. Kotz, "Application memory isolation on ultra-low-power mcus," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds., USENIX Association, 2018, pp. 127–132. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/hardin.

[107] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Good motive but bad design: Why ARM MPU has become an outcast in embedded systems," *CoRR*, vol. abs/1908.03638, 2019. arXiv: 1908.03638. [Online]. Available: http://arxiv.org/abs/1908.03638.

[108] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_04B-2_Kim_paper.pdf.

[109] B. Group, *2018 embedded systems safety & security study*, 2018. [Online]. Available: https://barrgroup.com/embedded-systems/surveys/2018.

[110] G. P. Zero, *Over the air: Exploiting broadcom's wi-fi stack*, https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, [Online; accessed 15-Feb-2020], 2019.

[111] axi0mX, *Ipwndfu*, https://github.com/axi0mX/ipwndfu, [Online; accessed 15-June-2020].

[112] D. Goodin, *This thumbdrive hacks computers. "badusb" exploit makes devices turn "evil"*, https://arstechnica.com/information-technology/2014/07/this-thumbdrive-hacks-computers-badusb-exploit-makes-devices-turn-evil/, [Online; accessed 15-June-2020], 2014.

[113] C. Herley and P. C. van Oorschot, "Sok: Science, security and the elusive goal of security as a scientific pursuit," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, IEEE Computer Society, 2017, pp. 99–120. DOI: 10.1109/SP.2017.38. [Online]. Available: https://doi.org/10.1109/SP.2017.38.

[114] J. Giraldo, D. I. Urbina, A. Cardenas, J. Valente, M. A. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Comput. Surv.*, vol. 51:no. 4, 76:1–76:36, 2018. DOI: 10.1145/3203245. [Online]. Available: https://doi.org/10.1145/3203245.

[115] Y. Shoukry, P. Martin, Y. Yona, S. N. Diggavi, and M. B. Srivastava, "Pycra: Physical challenge-response authentication for active sensors under spoofing attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 1004–1015. DOI: 10.1145/2810103.2813679. [Online]. Available: https://doi.org/10.1145/2810103.2813679.

[116] R. Mitchell and I. Chen, "A survey of intrusion detection techniques for cyber-physical systems," *ACM Comput. Surv.*, vol. 46:no. 4, 55:1–55:29, 2013. DOI: 10.1145/2542049. [Online]. Available: https://doi.org/10.1145/2542049.

[117] D. Hadziosmanovic, R. Sommer, E. Zambon, and P. H. Hartel, "Through the eye of the PLC: semantic security monitoring for industrial processes," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, Eds., ACM, 2014, pp. 126–135. DOI: 10.1145/2664243.2664277. [Online]. Available: https://doi.org/10.1145/2664243.2664277.

[118] Y. Liu, M. K. Reiter, and P. Ning, "False data injection attacks against state estimation in electric power grids," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., ACM, 2009, pp. 21–32. DOI: 10.1145/1653662.1653666. [Online]. Available: https://doi.org/10.1145/1653662.1653666.

[119] D. Mashima and A. A. Cárdenas, "Evaluating electricity theft detectors in smart grid networks," in *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds., ser. Lecture Notes in Computer Science, vol. 7462, Springer, 2012, pp. 210–229. DOI: 10.1007/978-3-642-33338-5_11. [Online]. Available: https://doi.org/10.1007/978-3-642-33338-5_11.

[120] A. A. Cárdenas, S. Amin, Z. Lin, Y. Huang, C. Huang, and S. Sastry, "Attacks against process control systems: Risk assessment, detection, and response," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, Eds., ACM, 2011, pp. 355–366. [Online]. Available: https://dl.acm.org/citation.cfm?id=1966959.

[121] Y. Wang, Z. Xu, J. Zhang, L. Xu, H. Wang, and G. Gu, "SRID: state relation based intrusion detection for false data injection attacks in SCADA," in *Computer Security - ESORICS 2014 - 19th European*

Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II, M. Kutylowski and J. Vaidya, Eds., ser. Lecture Notes in Computer Science, vol. 8713, Springer, 2014, pp. 401–418. DOI: 10.1007/978-3-319-11212-1_23. [Online]. Available: https://doi.org/10.1007/978-3-319-11212-1_23.

[122] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, I. Lee, and G. J. Pappas, "Robustness of attack-resilient state estimators," in *ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS, Berlin, Germany, April 14-17, 2014*, IEEE Computer Society, 2014, pp. 163–174. DOI: 10.1109/ICCPS.2014.6843720. [Online]. Available: https://doi.org/10.1109/ICCPS.2014.6843720.

[123] M. S. Chong, M. Wakaiki, and J. P. Hespanha, "Observability of linear systems under adversarial attacks," in *American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015*, IEEE, 2015, pp. 2439–2444. DOI: 10.1109/ACC.2015.7171098. [Online]. Available: https://doi.org/10.1109/ACC.2015.7171098.

[124] Y. Shoukry, M. Chong, M. Wakaiki, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, J. P. Hespanha, and P. Tabuada, "Smt-based observer design for cyber-physical systems under sensor attacks," *TCPS*, vol. 2:no. 1, 5:1–5:27, 2018. DOI: 10.1145/3078621. [Online]. Available: https://doi.org/10.1145/3078621.

[125] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, IEEE Computer Society, 2017, pp. 289–303. DOI: 10.1109/SP.2017.37. [Online]. Available: https://doi.org/10.1109/SP.2017.37.

[126] D. Midi, M. Payer, and E. Bertino, "Memory safety for embedded devices with nescheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds., ACM, 2017, pp. 127–139. DOI: 10.1145/3052973.3053014. [Online]. Available: https://doi.org/10.1145/3052973.3053014.

[127] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: automatic compartments for embedded systems," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., USENIX Association, 2018, pp. 65–82. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/clements.

[128] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "Mrai: Securing embedded systems with return address integrity," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/murai-securing-embedded-systems-with-return-address-integrity/.

[129] J. Zhou, Y. Du, L. Ma, Z. Shen, J. Criswell, and R. J. Walls, "Silhouette: Efficient intra-address space isolation for protected shadow stacks on embedded systems," 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie.

[130] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, V. Atluri, B. Pfitzmann, and P. D. McDaniel, Eds., ACM, 2004, pp. 298–307. DOI: 10.1145/1030083.1030124. [Online]. Available: https://doi.org/10.1145/1030083.1030124.

[131] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, IEEE Computer Society, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13. [Online]. Available: https://doi.org/10.1109/SP.2013.13.

[132] D. Bradbury, *The true cost of rewrites*, 2018. [Online]. Available: https://8thlight.com/blog/doug-bradbury/2018/11/27/true-cost-rewrites.html.

[133] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *25th Annual Network and Distributed Sys-*

tem Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_01A-4_Muench_paper.pdf.

[134] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, USENIX Association, 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[135] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, IEEE Computer Society, 2014, pp. 276–291. DOI: 10.1109/SP.2014.25. [Online]. Available: https://doi.org/10.1109/SP.2014.25.

[136] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*, S. D. Gribble, Ed., USENIX, 2003. [Online]. Available: http://www.usenix.org/events/usits03/tech/oppenheimer.html.

[137] D. Evans, A. Nguyen-Tuong, and J. C. Knight, "Effectiveness of moving target defenses," in *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, ser. Advances in Information Security, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., vol. 54, Springer, 2011, pp. 29–48. DOI: 10.1007/978-1-4614-0977-9_2. [Online]. Available: https://doi.org/10.1007/978-1-4614-0977-9_2.

[138] T. Rains, M. Miller, and D. Weston, "Exploitation trends: From potential risk to actual risk," in *RSA*, 2015.

[139] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium*

on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013, IEEE Computer Society, 2013, pp. 574–588. DOI: 10.1109/SP.2013.45. [Online]. Available: https://doi.org/10.1109/SP.2013.45.

[140] D. I. Urbina, J. A. Giraldo, A. A. Cárdenas, N. O. Tippenhauer, J. Valente, M. A. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the impact of stealthy attacks on industrial control systems," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, 2016, pp. 1092–1105. DOI: 10.1145/2976749.2978388. [Online]. Available: https://doi.org/10.1145/2976749.2978388.

[141] D. Liberzon, *Switching in Systems and Control*, ser. Systems & Control: Foundations & Applications. Birkhäuser, 2003, ISBN: 978-1-4612-6574-0. DOI: 10.1007/978-1-4612-0017-8. [Online]. Available: https://doi.org/10.1007/978-1-4612-0017-8.

[142] H. Lin and P. J. Antsaklis, "Stability and stabilizability of switched linear systems: A survey of recent results," *IEEE Trans. Automat. Contr.*, vol. 54:no. 2, pp. 308–322, 2009. DOI: 10.1109/TAC.2008.2012009. [Online]. Available: https://doi.org/10.1109/TAC.2008.2012009.

[143] G. Zhai, B. Hu, K. Yasuda, and A. N. Michel, "Stability analysis of switched systems with stable and unstable subsystems: An average dwell time approach," *Int. J. Systems Science*, vol. 32:no. 8, pp. 1055–1061, 2001. DOI: 10.1080/00207720116692. [Online]. Available: https://doi.org/10.1080/00207720116692.

[144] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 13th. Prentice-Hall, Inc., USA, 2017, ISBN: 0134407628.

[145] *Rusefi*. [Online]. Available: http://rusefi.com/wiki/index.php?title=Main_Page.

[146] *Chibios*. [Online]. Available: https://www.chibios.org/dokuwiki/doku.php.

[147] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *22nd Annual Network and Distributed Sys-*

tem Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015, The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/isomeron-code-randomization-resilient-just-time-return-oriented-programming.

[148] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, IEEE, 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074. [Online]. Available: https://doi.org/10.1109/ICRA.2015.7140074.

[149] *Nuttx*. [Online]. Available: https://nuttx.apache.org/.

[150] "Alert service bulletin b787-81205-sb270040-00," *Boeing Aircraft Service Bulletin*, 2016.

[151] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - A technique for cheap recovery," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, E. A. Brewer and P. Chen, Eds., USENIX Association, 2004, pp. 31–44. [Online]. Available: http://www.usenix.org/events/osdi04/tech/candea.html.

[152] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Digest of Papers: FTCS-25, The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, California, USA, June 27-30, 1995*, IEEE Computer Society, 1995, pp. 381–390. DOI: 10.1109/FTCS.1995.466961. [Online]. Available: https://doi.org/10.1109/FTCS.1995.466961.

[153] *SIGSOFT '83: Proceedings of the Symposium on High-Level Debugging*, Association for Computing Machinery, Pacific Grove, California, 1983, ISBN: 0897911113.

[154] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 10:no. 1, 8:1–8:34, 2014. DOI: 10.1145/2539117. [Online]. Available: https://doi.org/10.1145/2539117.

[155] K. M. M. Aung and J. S. Park, "Software rejuvenation approach to security engineering," in *Computational Science and Its Applications - ICCSA 2004, International Conference, Assisi, Italy, May 14-17, 2004, Proceedings, Part IV*, A. Laganà, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi, Eds., ser. Lecture Notes in Computer Science, vol. 3046, Springer, 2004, pp. 574–583. DOI: 10.1007/978-3-540-24768-5_61. [Online]. Available: https://doi.org/10.1007/978-3-540-24768-5_61.

[156] C.-Y. Lee, K. M. Kavi, M. Gomathisankaran, and P. Kamongi, "Security through software rejuvenation," in *The Ninth International Conference on Software Engineering Advances (ICSEA)," IARIA*, 2014, pp. 347–353.

[157] F. Abdi, C. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Guaranteed physical security with restart-based design for cyber-physical systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, C. Gill, B. Sinopoli, X. Liu, and P. Tabuada, Eds., IEEE Computer Society / ACM, 2018, pp. 10–21. DOI: 10.1109/ICCPS.2018.00010. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00010.

[158] R. Romagnoli, B. H. Krogh, and B. Sinopoli, "Design of software rejuvenation for CPS security using invariant sets," in *2019 American Control Conference, ACC 2019, Philadelphia, PA, USA, July 10-12, 2019*, IEEE, 2019, pp. 3740–3745. [Online]. Available: http://ieeexplore.ieee.org/document/8815155.

[159] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20:no. 4, pp. 398–461, 2002. DOI: 10.1145/571637.571640. [Online]. Available: https://doi.org/10.1145/571637.571640.

[160] J. S. Mertoguno, R. M. Craven, M. S. Mickelson, and D. P. Koller, "A physics⊠based strategy for cyber resilience of CPS," in *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, M. C. Dudzik and J. C. Ricklin, Eds., International Society for Optics and Photonics, vol. 11009, SPIE, 2019, pp. 79 –90. DOI: 10.1117/12.2517604. [Online]. Available: https://doi.org/10.1117/12.2517604.

[161] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987, ISBN: 0-201-10715-5. [Online]. Available: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx.

[162] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*, J. Chase and S. Seshan, Eds., USENIX, 2007, pp. 323–336. [Online]. Available: http://www.usenix.org/events/usenix07/tech/laadan.html.

[163] H. Nam, J. Kim, S. J. Hong, and S. Lee, "Secure checkpointing," *Journal of Systems Architecture*, vol. 48:no. 8, pp. 237 –254, 2003, ISSN: 1383-7621. DOI: https://doi.org/10.1016/S1383-7621(02)00137-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762102001376.

[164] S. Volckaert, B. Coppens, B. D. Sutter, K. D. Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, 2017, pp. 270–285. DOI: 10.1145/3064176.3064178. [Online]. Available: https://doi.org/10.1145/3064176.3064178.

[165] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "Kmvx: Detecting kernel information leaks with multi-variant execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 559–572. DOI: 10.1145/3297858.3304054. [Online]. Available: https://doi.org/10.1145/3297858.3304054.

[166] B. Cox and D. Evans, "N-variant systems: A secretless framework for security through diversity," in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, A. D. Keromytis, Ed., USENIX Association, 2006. [Online]. Available: https://www.usenix.org/

conference/15th-usenix-security-symposium/n-variant-systems-secretless-framework-security-through.

[167] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Second Edition*, 2nd ed., A. Oram, Ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[168] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 941–951. DOI: 10.1145/2810103.2813676. [Online]. Available: https://doi.org/10.1145/2810103.2813676.

[169] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 745–762. DOI: 10.1109/SP.2015.51. [Online]. Available: https://doi.org/10.1109/SP.2015.51.

[170] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. M. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 469–484. DOI: 10.1145/3297858.3304037. [Online]. Available: https://doi.org/10.1145/3297858.3304037.

[171] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, IEEE Computer Society, 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853210.

[172] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer Society, 2016, pp. 969–986. DOI: 10.1109/SP.2016.62. [Online]. Available: https://doi.org/10.1109/SP.2016.62.

[173] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, 2018, pp. 1868–1882. DOI: 10.1145/3243734.3243739. [Online]. Available: https://doi.org/10.1145/3243734.3243739.

[174] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50:no. 1, 16:1–16:33, 2017. DOI: 10.1145/3054924. [Online]. Available: https://doi.org/10.1145/3054924.

[175] C. Cowan, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, A. D. Rubin, Ed., USENIX Association, 1998. [Online]. Available: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[176] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured Programming*. Academic Press Ltd., London, UK, 1972, ISBN: 0-12-200550-3.

[177] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313. [Online]. Available: https://doi.org/10.1145/1315245.1315313.

[178] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of the 2008 ACM Conference on Computer*

and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008, P. Ning, P. F. Syverson, and S. Jha, Eds., ACM, 2008, pp. 27–38. DOI: 10.1145/1455770.1455776. [Online]. Available: https://doi.org/10.1145/1455770.1455776.

[179] U. Drepper, *Security enhancements in redhat enterprise Linux (beside SELinux)*, 2004.

[180] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, Eds., ACM, 2011, pp. 30–40. DOI: 10.1145/1966913.1966919. [Online]. Available: https://doi.org/10.1145/1966913.1966919.

[181] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, IEEE Computer Society, 2014, pp. 575–589. DOI: 10.1109/SP.2014.43. [Online]. Available: https://doi.org/10.1109/SP.2014.43.

[182] I. Qualcomm Technologies, *Pointer authentication on ARMv8.3*, https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, [Online; accessed 15-June-2020], 2017.

[183] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, IEEE Computer Society, 2012, pp. 601–615. DOI: 10.1109/SP.2012.41. [Online]. Available: https://doi.org/10.1109/SP.2012.41.

[184] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, K. Chen, Q. Xie, W. Qiu, N. Li, and W. Tzeng, Eds., ACM, 2013, pp. 299–310. DOI: 10.1145/2484313.2484351. [Online]. Available: https://doi.org/10.1145/2484313.2484351.

[185] LLVM, *Control flow integrity design*, https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, [Online; accessed 15-June-2020].

[186] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "Vtrust: Regaining trust on virtual calls," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/vtrust-regaining-trust-virtual-calls.pdf.

[187] D. Bounov, R. G. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/protecting-cpp-dynamic-dispatch-through-vtable-interleaving.pdf.

[188] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object type integrity for C++ virtual dispatch," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, ser. NDSS '18, San Diego, CA, USA, 2018.

[189] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard™: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, USENIX Association, 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/pointguard\%E2\%84\%A2-protecting-pointers-buffer-overflow.

[190] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, IEEE Computer Society, 2004, pp. 209–220. DOI: 10.1109/MICRO.2004.20. [Online]. Available: https://doi.org/10.1109/MICRO.2004.20.

[191] R. Avanzi, "The QARMA block cipher family - almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search

heuristics for low-latency s-boxes," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 444, 2016. [Online]. Available: http://eprint.iacr.org/2016/444.

[192] S. Schirra, *Ropper*, https://github.com/sashs/Ropper, [Online; accessed 15-June-2020].

[193] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: runtime intrusion prevention evaluator," in *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, R. H. Zakon, J. P. McDermott, and M. E. Locasto, Eds., ACM, 2011, pp. 41–50. DOI: 10.1145/2076732.2076739. [Online]. Available: https://doi.org/10.1145/2076732.2076739.

[194] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/ropecker-generic-and-practical-approach-defending-against-rop-attacks.

[195] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, IEEE Computer Society, 2014, pp. 227–242. DOI: 10.1109/SP.2014.22. [Online]. Available: https://doi.org/10.1109/SP.2014.22.

[196] Solar Designer, *Getting around non-executable stack (and fix)*, http://seclists.org/bugtraq/1997/Aug/63, 1997.

[197] Nergal, *The advanced return-into-lib(c) exploits: PaX case study*, http://phrack.org/issues/58/4.html, [Online; accessed 15-June-2020], 2001.

[198] Exploit Database, *Mcrypt 2.5.8 - local stack overflow*, https://www.exploit-db.com/exploits/22928, [Online; accessed 15-June-2020].

[199] ——, *Nginx 1.3.9 < 1.4.0 - chuncked encoding stack buffer overflow*, https://www.exploit-db.com/exploits/25775, [Online; accessed 15-June-2020].

[200] ——, *Apache 2.4.7 + php 7.0.2 - openssl_seal() uninitialized memory code execution*, https://www.exploit-db.com/exploits/40142, [Online; accessed 15-June-2020].

[201] ——, *Netperf 2.6.0 - stack-based buffer overflow*, https://www.exploit-db.com/exploits/46997, [Online; accessed 15-June-2020].

[202] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39:no. 2, pp. 1–7, 2011. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[203] J. Bucek, K. Lange, and J. von Kistowski, "SPEC CPU2017: next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, K. Wolter, W. J. Knottenbelt, A. van Hoorn, and M. Nambiar, Eds., ACM, 2018, pp. 41–42. DOI: 10.1145/3185768.3185771. [Online]. Available: https://doi.org/10.1145/3185768.3185771.

[204] musl, *Musl libc*, http://www.musl-libc.org/, [Online; accessed 15-June-2020].

[205] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "Revarm: A platform-agnostic ARM binary rewriter for security applications," in *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, ACM, 2017, pp. 412–424. DOI: 10.1145/3134600.3134627. [Online]. Available: https://doi.org/10.1145/3134600.3134627.

[206] D. Ha, W. Jin, and H. Oh, "REPICA: rewriting position independent code of ARM," *IEEE Access*, vol. 6, pp. 50 488–50 509, 2018. DOI: 10.1109/ACCESS.2018.2868411. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2868411.

[207] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*

[ASPLOS 2020 was canceled because of COVID-19], J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, 2020, pp. 133–147. DOI: 10.1145/3373376.3378470. [Online]. Available: https://doi.org/10.1145/3373376.3378470.

[208] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, M. I. Schwartzbach and T. Ball, Eds., ACM, 2006, pp. 158–168. DOI: 10.1145/1133981.1134000. [Online]. Available: https://doi.org/10.1145/1133981.1134000.

[209] S. Volckaert, B. Coppens, and B. D. Sutter, "Cloning your gadgets: Complete ROP attack immunity with multi-variant execution," *IEEE Trans. Dependable Secur. Comput.*, vol. 13:no. 4, pp. 437–450, 2016. DOI: 10.1109/TDSC.2015.2411254. [Online]. Available: https://doi.org/10.1109/TDSC.2015.2411254.

[210] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds., USENIX Association, 2016, pp. 167–179. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert.

[211] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, IEEE Computer Society, 2016, pp. 431–442. DOI: 10.1109/DSN.2016.46. [Online]. Available: https://doi.org/10.1109/DSN.2016.46.

[212] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "LDX: causality inference by lightweight dual execution," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA,*

USA, April 2-6, 2016, T. Conte and Y. Zhou, Eds., ACM, 2016, pp. 503–515. DOI: 10.1145/2872362.2872395. [Online]. Available: https://doi.org/10.1145/2872362.2872395.

[213] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., ser. Lecture Notes in Computer Science, vol. 9721, Springer, 2016, pp. 322–342. DOI: 10.1007/978-3-319-40667-1_16. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_16.

[214] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee, "Stopping memory disclosures via diversification and replicated execution," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[215] T. Nyman, J. Ekberg, L. Davi, and N. Asokan, "CFI care: Hardware-supported call and return enforcement for commercial microcontrollers," in *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., ser. Lecture Notes in Computer Science, vol. 10453, Springer, 2017, pp. 259–284. DOI: 10.1007/978-3-319-66332-6_12. [Online]. Available: https://doi.org/10.1007/978-3-319-66332-6_12.

[216] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., USENIX Association, 2016, pp. 367–382. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king.

[217] A. Kwon, U. Dhawan, J. M. Smith, T. F. K. Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM, 2013, pp. 721–732.

DOI: 10.1145/2508859.2516713. [Online]. Available: https://doi.org/10.1145/2508859.2516713.

[218] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9. [Online]. Available: https://doi.org/10.1109/SP.2015.9.

[219] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, M. Annavaram, T. M. Pinkston, and B. Falsafi, Eds., IEEE Computer Society, 2018, pp. 600–611. DOI: 10.1109/ISCA.2018.00056. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00056.

[220] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, ACM, 2019, pp. 558–571. DOI: 10.1145/3352460.3358299. [Online]. Available: https://doi.org/10.1145/3352460.3358299.

[221] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010. [Online]. Available: https://doi.org/10.1109/SP.2019.00010.

[222] *CVE–2017–5115*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5115, [Online; accessed 30-Aug-2019], 2017.

[223] *CVE–2014–1444*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1444, [Online; accessed 30-Aug-2019], 2014.

[224] Y. Jeon, P. Biswas, S. A. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for C++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 2373–2387. DOI: 10.1145/3133956.3134062. [Online]. Available: https://doi.org/10.1145/3133956.3134062.

[225] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, 2016, pp. 920–932. DOI: 10.1145/2976749.2978366. [Online]. Available: https://doi.org/10.1145/2976749.2978366.

[226] G. J. Duck and R. H. C. Yap, "Effectivesan: Type and memory error detection using dynamically typed C/C++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds., ACM, 2018, pp. 181–195. DOI: 10.1145/3192366.3192388. [Online]. Available: https://doi.org/10.1145/3192366.3192388.

[227] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018. arXiv: 1802.09517. [Online]. Available: http://arxiv.org/abs/1802.09517.

[228] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, S. J. Eggers and J. R. Larus, Eds., ACM, 2008, pp. 103–114. DOI: 10.1145/1346281.1346295. [Online]. Available: https://doi.org/10.1145/1346281.1346295.

[229] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN,*

USA, June 14-18, 2014, IEEE Computer Society, 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853201.

[230] J. Woodruff, A. Joannou, H. Xia, A. C. J. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Trans. Computers*, vol. 68:no. 10, pp. 1455–1469, 2019. DOI: 10.1109/TC.2019.2914037. [Online]. Available: https://doi.org/10.1109/TC.2019.2914037.

[231] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. K. Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds., ACM, 2015, pp. 487–502. DOI: 10.1145/2694344.2694383. [Online]. Available: https://doi.org/10.1145/2694344.2694383.

[232] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, IEEE Computer Society, 2012, pp. 189–200. DOI: 10.1109/ISCA.2012.6237017. [Online]. Available: https://doi.org/10.1109/ISCA.2012.6237017.

[233] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, D. R. Kaeli and T. Moseley, Eds., ACM, 2014, p. 175. [Online]. Available: https://dl.acm.org/citation.cfm?id=2544147.

[234] T. Zhang, D. Lee, and C. Jung, "BOGO: buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17,*

2019, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 631–644. DOI: 10.1145/3297858.3304017. [Online]. Available: https://doi.org/10.1145/3297858.3304017.

[235] Oracle, *Hardware-assisted checking using Silicon Secured Memory (SSM)*, https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html, 2015.

[236] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, IEEE Computer Society, 2005, pp. 291–302. DOI: 10.1109/HPCA.2005.29. [Online]. Available: https://doi.org/10.1109/HPCA.2005.29.

[237] D. Weston and M. Miller, *Windows 10 mitigation improvements*, Black Hat USA, 2016.

[238] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. [Online]. Available: https://doi.org/10.1109/SP.2019.00002.

[239] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *NDSS '17: Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[240] K. Cook, *Introduce struct layout randomization plugin*, https://lkml.org/lkml/2017/5/26/558, 2017.

[241] N. Hussein, *Randomizing structure layout*, https://lwn.net/Articles/722293/, 2017.

[242] L. Eeckhout, *Computer architecture performance evaluation methods*, 1st. 2010.

[243] L. K. John, "More on finding a single number to indicate overall performance of a benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 32:no. 1, pp. 3–8, 2004.

[244] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds., USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[245] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. D. Son, and J. Woodruff, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042. [Online]. Available: https://doi.org/10.1145/3297858.3304042.

[246] J. Shi, Q. Long, L. Gao, M. A. Rothman, and V. J. Zimmer, *Methods and apparatus to protect memory from buffer overflow and/or underflow*, International patent WO/2018/176339, 2018.