YOLO: A New Security Architecture for Cyber Physical Systems

ABSTRACT

Cyber-physical systems (CPS) are defined by their unique characteristics involving both the cyber and physical domains. Their hybrid nature introduces new attack vectors but also provides an opportunity to design of new security architectures. In this work, we present *YOLO*,— You Only Live Once — a security architecture that leverages two unique physical properties of a CPS, inertia: the tendency of objects to stay at rest or in motion, and its built-in reliability to intermittent faults to survive CPS attacks.

At a high level, YOLO aims to use a new diversified variant for every new sensor input to the CPS'. The delays involved in YOLO, viz., the delays for rebooting and diversification, are easily absorbed by the CPS because of the inherent inertia and their ability to withstand minor perturbations. We implement YOLO on an open source Car Engine Control Unit, and with measurements from a real race car engine show that YOLO is imminently practical.

CCS Concepts

•**Security and privacy → Domain-specific security and privacy architectures;** *Embedded systems security;*

Keywords

CPS, security, diversification, reboot

1. INTRODUCTION

Cyber-physical systems (CPS) represent the synthesis of computational and physical processes encompassing a wide range of applications including transportation, medical devices, robots and power grids. CPS are defined by their unique characteristics involving feedback control loops with the physical world. The hybrid nature of CPS introduces new attack vectors that encompass both the physical and cyber realms, leading to a number of incidents in recent years [25, 1, 2, 38]. The goal of these attacks is to elicit incorrect and potentially dangerous behavior by compromising the physical operation of the system.

In this paper, we propose a new security architecture that leverages unique properties of CPS to assure secure operation for CPS controllers. The YOLO (You Only Live Once) principle aims to provide this assurance by limiting the duration and attack surface of a system. A key innovation in our approach is that we take advantage of physical properties of CPS such as inertia and its built-in reliability to intermittent faults to survive attacks. YOLO is best explained with an example. Consider a car: even if power is cut off to the car engines, the car will continue to run due to inertia; similarly, even if one or few sensor inputs are incorrect, the car will continue to work correctly because intermittent sensor errors happen in nature and controllers are designed to handle this case. In YOLO, we take advantage of these features. We intentionally reboot the system periodically to clear "tainted" state left by an attacker. During reboots, we rely on system inertia for the system to continue working. To mitigate vulnerabilities between reboots, we rely on diversification to force the attacker to develop a new attack strategy for every input.

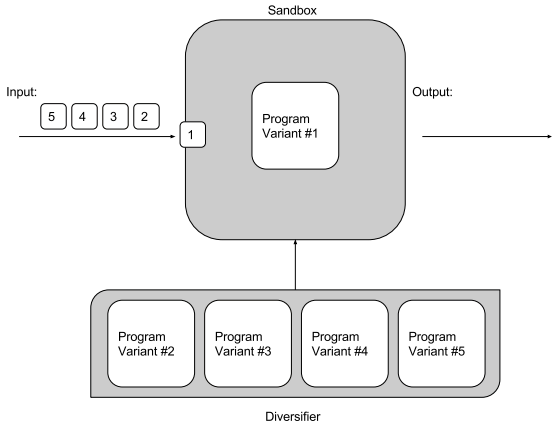


Figure 1: YOLO Overview. A security architecture for CPS involving reboot and diversification.

Figure 1 gives an idealized overview of this concept. In this context, a YOLO-ized controller is a stateless, event driven program. At time t_0 a particular program variant is loaded into a sandbox where it waits for an input. At time t_1 an input arrives where program variant is used to process it. Once the input has been processed, its output leaves the sandbox, the sandbox is reinitialized, i.e., rebooted, and a second program variant is loaded. This continues throughout the lifetime of the controller. Each sensor input and each variant are only allowed to effect the control system over a well defined life span, thus the YOLO acronym. Any attacker that succeeds in gaining control over the sandboxed program will only have control until some expected runtime of the system expires, and the program is replaced by a new variant.

The challenge in realizing this model in CPS is that most

systems have feedback loops that operate on prior state. Constant and naive rebooting at unexpected times may cause errors in the control algorithm. Further the computational resources are fairly constrained compared to traditional systems. In this paper we show how these challenges can be mitigated to capture the benefits of YOLO in pragmatic ways. We make two key contributions that allows us to overcome these challenges: first, we observe that even though most control systems are closed loop systems, for the brief period during reboot they can be considered as open loop systems. During this period we rely on inertia to overcome incorrect outputs from the control algorithm. Second, we show how to change the state estimation routines in CPS to account for tainted state. The basic idea here is to estimate the state using a sliding window that contains all but the latest input to estimate state. While this creates a small error in state estimation, the effect is similar to a single faulty sensor reading that most CPS control algorithms are already designed to handle. Thus we "piggy-back" on existing safety mechanisms to mitigate security risks.

We evaluate the YOLO concept on an Engine Control Unit (ECU) of a car. Using an open source engine controller (rusEFI), and measurements from a real car engine, we first discuss how existing code can be refactored with small changes to take advantage of the YOLO model, and then measure the performance and safety impacts of the YOLO model. We show that with our implementation and optimizations the two key costs of YOLO: the cost of reboots and the cost of diversification can be hidden from user. Specifically, we show that for our car engine, rebooting every 8th revolution at 4500 RPM does not incur any overheads, and that between reboots the system can be protected with extant diversification techniques such as control-flow diversification without missing any deadlines.

To summarize the main contributions of the paper are:

(1) An observation that Cyber Physical System have unique properties such as system inertia and resilience to intermittent faults that can be leveraged to improve security.

(2) A new security architecture called YOLO that combines periodic rebooting and diversification to take advantage of the unique properties of CPS'. As such YOLO is agnostic to specific rebooting or diversification techniques as long as they do not impact the safety of the system.

(3) Mechanisms to optimize two aspects of YOLO, rebooting and diversification, in the context of Cyber Physical Systems. While both rebooting and diversification are known techniques, the main novelty is the using them in combination in the context of CPS. We develop micro-rebooting techniques and low resource diversification techniques that are suitable for CPS'.

(4) Implementation and measurement of YOLO on an open source Car Engine to demonstrate the challenges and feasibility of YOLO.

The rest of the paper is organized as follows: In Section 2 of the paper we provide an overview of the system and threat model; then in the way of background we discuss the unique properties of CPS' in Section 3. In Section 4, we discuss the YOLO architecture and specific optimizations for rebooting and diversification. In Section 5 we describe a YOLO-ized Engine Control Unit and in Section 6 we describe the results of our evaluation. Section 7 describes related work and we conclude in Section 8.

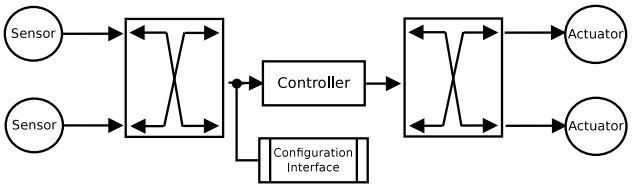


Figure 2: CPS System Model. A minimalist view of a cyber-physical system.

2. SYSTEM AND THREAT MODEL

Like any other system, security in CPS' is a full-system property. This means that all aspects of the system including its configuration, construction and operation should be secure for the system to be secure. A CPS, broadly speaking, has three attack surfaces: the CPS' interface to the physical world through sensors and actuators, the controller and all software running on it, and the network that connects the controller to the sensors and actuators (See Figure 2).

In this paper we consider a minimalist CPS model. This means that the ECU we will evaluate is devoid of "bells and whistles" software such as a Bluetooth stack, CD player, an open Wifi port etc. Said differently, in our model we assume that the hardware and software functionality available in the system is solely that required to support correct control over the physical plant of the system. The rationale for this choice is to focus our attention on aspects that are unique to CPS.

The minimalist CPS offers attackers a rich set of attack vectors. As can be seen from the canonical CPS reference figure, the attackers can spoof/corrupt sensor inputs, or attack the network connection between the controller and the sensor, say through debug ports, or overwrite the configuration state and/or the control algorithm using some physical access.

We make the following assumptions about adversarial capabilities:

- The adversary's main objective is to inhibit correct operation of the physical process being controlled.
- The adversary has access to the source code of the system and/or the underlying control algorithms being used. This is a strong assumption about adversarial capabilities.
- We assume the adversary can only spoof input signals for some bounded duration until correct signals are again received. Sensor data authentication techniques, such as GPS authentication, or physically distributed redundant sensors [7], can be used to provide this guarantee.
- The controller software is not bug-free. The exact nature of the bugs is not really important to our work except we assume that at least one of these bugs can be exploited through bad inputs either from spoofed sensors or network injection messages. For instance, a bad sensor input may result in a integer overflow vulnerability that be escalated to number of different vulnerabilities.

With these assumptions in mind, many forms of classical attack vectors are available. For example, in an ECU,

certain engine configuration parameters are application and engine specific. The car is tuned by empirical observation, and therefore even a minimal engine controller must support reconfiguration over an external interface, which is usually accessible through a network. Such an interface presents an opportunity for both remote or local exploits.

3. UNIQUE CPS PROPERTIES

Cyber-physical systems have important properties that allow for interesting security techniques to be explored. These properties can be divided into two categories, one influenced by the physical requirements and the other by software engineering requirements.

System Properties: The first is the physical property of inertia. It is the resistance of an object to any change in its motion. This principle is essential as it asserts that the physical components of the system should continue operating in some state without any external forces. In fact, physical systems are sometimes engineered to take advantage of this property. One early example of such a system was the hit-and-miss engine [27] in which the engine fires and the coasts for some time and fires again to maintain its average speed. Today, high performance cars will allow the driver temporarily disable fuel ignition in order to shift gears quickly without depressing the accelerator [40].

The second property is that of resilience in the control algorithms. Many CPS are expected to perform in situations where their inputs are subject to unavoidable sources of environmental noise and interference. Thus, algorithms are designed to tolerate certain amounts of error and still function correctly. While this tolerance may not necessarily provide robustness against a malicious attacker, it allows for defenses that can exploit this robustness. These defenses could potentially invalidate some aspects of the running state under the assumption that there is some level of tolerance to input noise.

Software Architecture: CPS systems are typically structured as event driven programs. The control flow of the program follows sensor updates, triggering the system to calculate new state estimates derived from that data which then affect its behavior. This programming style lends itself to being more easily restructured and modeled.

CPS controller software typically requires state history to estimate observations of the physical environment. We use the following taxonomy to describe each type of state.

1. *Application Configuration State* - State which is once set is rarely set again. Examples include controller gains and user defined constants.

2. *Hardware Configuration State* - State that can differ from what was originally configured to what is currently set in the hardware. Examples include the privilege level or the clock source of the processor, and hardware peripherals.

3. *Cached Event State* - State which records sensor inputs. Examples include buffered analog inputs and buffered communication channels.

4. *Multi-period State* - State that is estimated across multiple periods of the system. Examples include speed and acceleration estimations.

At a minimum, a cyber-physical system should be expected to have statefulness of category 1 and 2. Any feedback loops in the system imply the existence of category 3 and 4.

4. YOLO SECURITY ARCHITECTURE

When considering any system, especially those that are updated rarely such as CPS, it seems impossible to build a defense that can protect against all possible future intrusions. Instead, YOLO aims to prevent persistent threats from establishing a stronghold on the system. It does so by emphasizing recoverability methods which attempt to restore the system to a well-modeled state.

YOLO takes advantage of two orthogonal, but complementary security techniques: reboot and diversification. In combination with the inherent inertial properties of CPS, these two techniques can be used to construct an ideal environment where a particular diversified program is used once to process an incoming input before another variant is used. YOLO asserts that any input must have a bounded time horizon over which it can affect the system. Ideally, any exploitable subsystem only affects the system for the minimum possible time before being terminated, replaced, or reinitialized. Additionally, no single exploit should succeed on a particular subsystem more than once.

There are two key properties of CPS that make the task of YOLO-izing a system particularly difficult: statefulness and the observability of the physical state. Many control algorithms require state history to estimate certain values. This state history is even more important when it is used to bridge the gap between program state and physical properties that can only be observed intermittently, such as switch or clock signals. These two aspects require that YOLO be able to maintain or re-synchronize consistency with the physical environment to ensure correct behavior.

4.1 Rebootability

Why reboot?: Even among expert users, rebooting is the prefered solution for nearly any problem in the computing world. It is a simple and universally applicable panacea for software problems. The simple intuition behind the unreasonable effectiveness of rebooting is that the software is tested most often in its pristine, freshly rebooted state [31, 14].

From the point of view of thwarting an attacker, the restoration of state typically involved with a reboot helps prevent an attacker's ability to gain a persistent method of execution. In a reboot, important hardware parameters such as core registers and peripheral configurations that define things such as interrupts, are brought back to a default value. At any later phase in execution, the combinatoric explosion of potential states makes validation more difficult. The conditions of the reboot sequence provide predictable and well defined.

Although realistic CPS require stateful, closed feedback loops, *we observe that due to CPS properties such as inertia, one can operate as an open feedback system for a bounded period of time.* This allows us to reboot the system and return to a well defined state when a particular piece of state becomes corrupted.

Cost: In most situations, simple rebooting can incur a high penalty especially in the context of a cyber-physical system degrading optimal performance. Several factors can

contribute to the high overhead, the first being the downtime the chip requires to effect a reboot. The second involves the default values taken by peripheral devices which may have unintended physical consequences. The last factor involves the efficiency of the startup routines and warm-up times of certain functionality.

Unlike traditional computing environments, where rebooting occurs at the second time scale, rebooting times for microcontrollers and CPS software is an order of magnitude faster occurring at the millisecond scale. Additionally, the physical components typically controlled occur at human time scales which allow us to tolerate the reboot times. However, to reduce this cost and achieve closer to optimal performance, we define a layered approach of micro-reboots where each consecutive layer is more expensive and intrusive. Micro-reboots involve the individual rebooting of fine-grained application components, commonly known as microservices, and have been previously explored in the context of web applications [8] and low-level system software [24].

Optimization: YOLO's overarching strategy of micro-reboot layers is to explicitly attempt to forget a given input as quickly as possible, limiting the effects of any malicious input. We accomplish this by recalculating the system state as though that input had not been observed. The control algorithms can tolerate this missed input as if it were noise, leading to imprecise estimations that still allow us to continue operation, if however sub-optimally. If this cannot be done, the state may be reset to some default value and all micro-services which depend on the discarded values are recursively recomputed. Unlike micro-reboots discussed in Candea et al. [8], when all else fails, we reboot the hardware platform and rely on inertia to allow recovery without catastrophic failure.

4.2 Diversification

Why diversify?: The goal of YOLO is to increase our confidence that the system is functioning as designed the majority of the time. The principle method we have discussed so far for assuring this condition is periodic microreboot actions that return the system to a well-modeled state. This opens a vulnerability window in which an attacker can exploit the system between microreboots. Diversification lowers the likelihood that an attacker can successfully exploit the system between reboots. Paired with rebootability, highlights the tradeoff between the integrity and performance of the system. Further with diversification, YOLO can perform these microreboot actions less often, reducing the performance penalties associated with them.

Cost: The YOLO paradigm is agnostic to the diversification technique. However, the additional delays imposed by these techniques should not affect the real-time deadlines of the CPS. These overheads vary from strategy to strategy, but are usually the result of encryption/decryption, random number generators, and additional read/writes required for their implementations. By leveraging the security that diversification provides while the program is running, we study the performance tradeoff varying system uptime and reboot frequency. These results are discussed in our evaluation (See Section 7).

Optimization: YOLO can mitigate the impact of diversification by performing computation tasks as background jobs. However, the complexity incurred by delegating diversification tasks to background jobs is unnecessary as we observe that typical delays imposed are significantly smaller when compared to the real-time deadlines of the physical subsystem.

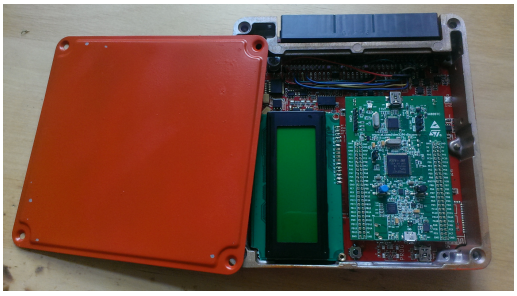


Figure 3: rusEFI ECU. The opensource platform with the STM32F4 Discovery board.

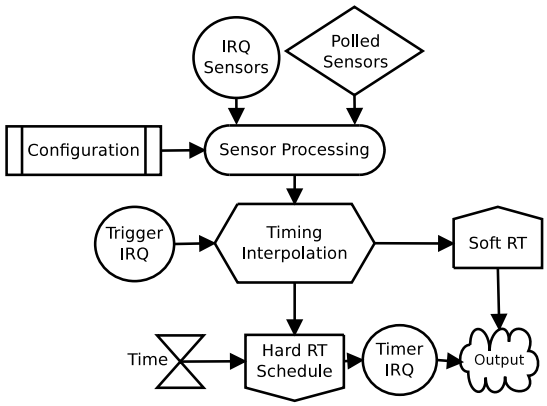


Figure 4: ECU Overview. The software architecture used to model the engine control unit.

5. YOLO-IZING AN ECU

We implement the YOLO paradigm on an engine control unit (ECU). The ECU is an often used system representative of many CPS as it realizes a broad cross section of the challenges that make them different from traditional computer systems. An ECU is the brain of an engine, designed to directly process inputs from a series of sensors and manage actuators to control the process of internal combustion.

As is common in CPS, an ECU must perform a set of real-time tasks to ensure proper engine functionality. For an engine to produce power, it must inject fuel into its internal chamber, mix it with air by controlling the timing of valves, and finally ignite the air-fuel mixture so that it combusts and rotate the shafts connected to the transmission. Typical engines perform these steps in what is called the four-stroke cycle. For the ECU to enable the actuators that control this process, it must be able to correctly decode the position of the engine with respect to the four-stroke cycle.

Baseline System: For our case study we use the rusEFI open-source ECU (See Figure 3) and a Honda CBR600RR engine, a very commonly hacked engine used by enthusiasts. rusEFI's main responsibilities include controlling fuel injectors, ignition, fuel pumps, and the valves as discussed above. The source-code is written in C/C++ running on top of

Table 1: ECU State Categorization

Application Configuration	Engine settings, Tuning parameters
Hardware Configuration	GPIO, Interrupt Vector Table
Cached Event	Sensor inputs: coolant temperature, airflow, etc.
Multi-Period	RPM, Engine Position Trigger

an open-source Real-Time OS (RTOS) called ChibiOS and is designed to run on a STM32F4-Discovery Board. This board contains a 168 MHz ARM Cortex-M4 processor with 192 Kbytes of SRAM and 1 MB of flash. As is typical for these devices the instruction fetching path is optimized for flash. Compared to flash instructions issued from the SRAM suffer a 50% performance penalty.

The Honda CBR600RR engine weighs around 130lbs and involves the rotation of various shafts along the engine. The inertia inherent in these rotations is crucial to our implementation of YOLO. In fact, certain shifting methods such as powershifting, take advantage of this property. Powershifting, involves cutting the injection and ignition, effectively allowing the engine to rotate freely as shifting completes.

The rusEFI ECU is structured as an event driven program shown in Figure 4. There are usually two types of sensors: polled and interrupt driven. Engine position events and certain ADC sensors generate interrupts that must be handled immediately to ensure engine operation. Other less critical sensors such as coolant temperature and engine air flow can be polled on demand as needed. Data from the sensors is then processed and used for state estimation. Control algorithms then schedule hard real-time tasks such as: injection and ignition, and soft real-time tasks such as reporting speed to the speedometer.

5.1 Implementing Reboots

We implement different strategies for effectively dealing with the different classes of state mentioned in Section 3. The goal of each layer is to enable partial reboots to help reduce the overhead compared to simple rebooting, but do so in a way that does not compromise the isolation that rebooting is meant to achieve. Table 1 gives examples of how certain state tracked by rusEFI can be categorized.

5.1.1 Application Configuration State

An adversary that manages to attack this state can corrupt the calibration parameters of engine temperature sensors causing it to overheat. To remedy this type of attack YOLO exploits the static nature of most configuration state, to perform validation. Engine tuning parameters and configuration are cryptographically signed when updated. The signature is validated against the current configuration periodically in non-realtime background threads. When configuration states differ from the expected value, a valid default is checked out from a secure store. The secure store in the STM32F4 is implemented by allocating a memory region protected by the MPU.

5.1.2 Hardware Configuration State

We observe that the default hardware state for different peripherals can trigger actuators at incorrect times. For example, consider a peripheral which is configured to control an actuator expecting a logic low to trigger and the default peripheral reboot state sets it to be a logic low. Under this

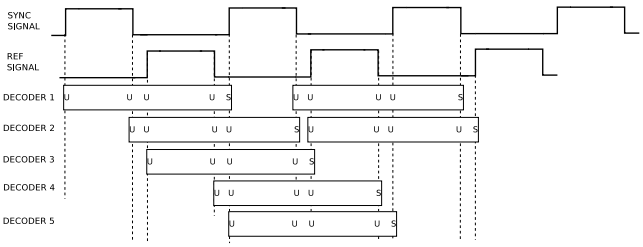


Figure 5: An illustration of the engine position decoder with replication. U and S, represent whether a decoder is unsynchronized or synchronized, respectively.

scenario, rebooting the peripheral can adversely affect the engine's optimal performance by issuing an injection or ignition event at inappropriate times. We alleviate these issues by triggering the peripheral reboot as a last resort. YOLO does so by implementing a device driver abstraction to maintain correct hardware state synchronization.

Our approach requires that each device driver contain three methods: validate, initialize, and reset. In the validate method we verify the consistency of device control registers. For example, we verify whether a GPIO pin is configured to be logic high. Initialize returns the device to a consistent state using the configuration state without resetting the hardware. This avoids inconsistencies where resetting an output pin can trigger incorrect ignition event timing. Finally, reset escalates to rebooting the hardware peripheral. The STM32F4 only allows us to reset GPIO banks made up of multiple ports. Therefore, all previously set configuration for all ports is lost causing us to reinitilialize all associated drivers.

We highlight our approach through an example. The STM32F4 contains a method to freeze the configuration of a GPIO bank which requires a reset to unlock its effects. If we assume an attacker that freezes the configuration of a bank after modifying an output to an input pin we follow the steps taken by our defense: the validation method will detect the incorrectly configured pin, which will escalate to the initialization method. The initialization method will then attempt to write the correct configuration to the bank, but fail because it is frozen. Finally, the reset method will then issue a hardware reset of the bank.

5.1.3 Cached Event State

From the perspective of an attacker cached state can be used to feed malicious inputs to other parts of the system. YOLO handles cached event state as non-authoritative, disposable state. Depending on the cached event in question, other microreboot layers can invalidate this state as necessary. If the cached data is invalidated, we simply poll the sensor again. Making this state non-authoritative limits the effects an attacker's corruption of these values could have.

5.1.4 Multi-period State

State estimation usually occurs through several consecutive observations of input data potentially across multiple periods. An attacker can exploit the time it takes to observe these events to force the ECU's engine position decoder to believe it is synchronized for an indeterminate amount of time. By allowing the ECU to believe it is synchronized, ignition and injection events can be incorrectly scheduled

which may cause physical harm to the engine. We take two approaches to ameliorating these attacks.

One approach involves replication similar to those used by other fault tolerant techniques that exploit the idea of consensus between untrusted observers [18, 32]. Typically, such systems use consensus testing among multiple observers in a distributed system to overcome some number of untrusted actors. In this case, our goal is not to come to a consensus on the individual messages, but on the state of the system given a time series of messages. We consider each message as potentially being compromised, and therefore test the consensus of the observers across a sliding window of received messages, discarding the old input as soon as it is feasible. This limits the lifespan of a compromised message to the duration of the minimum sequence of messages necessary to make an estimation. The number of replicated observer instantiations depends on the state being observed, as we wish to have a new estimate become available on every observed message. This approach is mainly appropriate when a state estimator which requires multiple messages over time is implemented in an object oriented manner, and we want to enable reboot of that estimator as often as possible without violating the encapsulation of that estimator. This strategy allows us to define a reboot for that object without a performance penalty.

The implementation of this strategy for an engine position decoder is demonstrated in Figure 5. For the engine position decoder, as long as one of the instantiations does not observe the attacker's input, it should not affect that instantiations estimation. This allows recovery of the correct estimate. For the engine position decoder, the shape of the signal determines the number of decoder replicants. Figure 5 illustrates a simplified signal shape observed by the decoders, where U and S correspond to when the decoder considers itself unsynchronized or synchronized. Any replicant which believes that it is synchronized must agree on its estimate of the state. At each time step, all replicants process the event signal. If at any point we detect any discrepancies in the decoder synchronization states, our system goes into a 'reboot' phase, where we consider ourselves to be unsynchronized until we have observed enough events that at least one replicant believes it is synchronized.

A second approach involves explicitly regenerating any state that does not require observations from the outside world to be reconstructed. One example would be constant recomputation of task schedules such as ignition and injection events. Specifically, for engine position events which regularly schedule these tasks, we periodically discard the existing one and recompute. This approach is similar to the rejuvenation discussed in [8]. By following this approach we prevent an attacker from compromising the controller by inserting their own task into the schedule.

5.2 Diversification

Embedded platforms typically found in cyber-physical systems, much like the STM32F4 used by rusEFI, have limited resources. These constraints restrict our choice of diversification methods to those which can be implemented efficiently on embedded devices. Performance is not the only restriction, the diversification strategies must also provide protection against memory vulnerabilities. Among the various strategies available, we focus on a small subset: LR^2 [6], Isomeron [12] and ISR [33, 36]. Each approach has various levels of runtime cost. We discuss this later in Section 7.

5.2.1 LR^2

LR^2 or Leakage-Resilient Layout Randomization, enforces execute-only-memory(XoM) in software. It makes use of hardware that can enforce (W⊕X) which is commonly provided by either an MPU or MMU. LR^2 divides the memory address space into two regions: code and data. It uses this division and load masking to enforce the property that load operations cannot access code pages, limiting the attackers ability to create ROP gadget chains. Our implementation follows the original closely.

5.2.2 Isomeron

Isomeron introduces a hybrid defense approach that combines code randomization with execution path randomization. The main security objective of Isomeron 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. One copy of the program, A is the original application code, while the other, B, is diversified using any fine-grained ASLR. This ensures that gadgets across both versions are at different addresses. Thus, since the attacker cannot predict when either A or B will be executed, they cannot construct a correct gadget chain.

The original implementation of Isomeron uses dynamic binary instrumentation techniques. This approach is not feasible on resource constrained devices, hence we implement Isomeron using static techniques. Leveraging existing `BinUtils` functionality and a custom binary rewriting tool, our implementation makes it suitable for resource constrained devices.

There are three major components to implementing Isomeron: program twinning, execution randomization, and function call instrumentation. We discuss each briefly to provide an overview of our static implementation. Program twinning is done in three steps: cloning, patching, and linking. For cloning, we first begin by separating out code from data using the appropriate compiler flags such as `-ffunction-sections` and `-fdata-sections`. Then, using the binary rewriter, we patch the data relocation information of version B to point to A. Finally, we have the linker stitch things up to create the final `ELF` file. Execution randomization is performed using a source of randomness, in our case a hardware random number generator (RNG). The values from the RNG are stored into a protected memory region and used to perform execution randomization at the granularity of a function call. The final component involves instrumenting function calls to allow for randomized execution paths. This step is performed using linker flags such as `-wrap` and shadow stacks, in order to encapsulate function entry and exit with function trampolines.

5.2.3 Isomeron + LR^2

LR^2 can be used to strengthen and improve Isomeron's performance, and Isomeron can be used for adding extra security benefits to LR^2. We implement a hybrid approach that leverages the features of both schemes. Using LR^2 to provide the foundation in order to handle memory protection mechanisms and Isomeron's execution path randomization to increase the entropy of the program, results in a diversification strategy with better performance than Isomeron

alone.

5.2.4 Instruction Set Randomization

Instruction Set Randomization (ISR) is a technique that mitigates attacks by encoding instructions. A simple way to accomplish this is to XOR every word in text pages offline with a unique key and use the same key again online to decode them just before execution.

Our ISR method uses an MPU to perform Just-In-Time decryption of the instruction stream, analogously to the implementation in [37]. The MPU provided in STM32F4 has eight regions. We use four of them to protect flash and SRAM and also to enforce W⊕X. We use the remaining four regions to set a work memory area for ISR on SRAM.

Each MPU region is set to cover 1KB since the maximum size of a function in rusEFI is less than 4KB. There are eight subregions for each MPU region allowing text pages to be decoded in 128B units. In general, the unit size is a design parameter and should be tuned for specific applications.

We use overlay and function wrapping features provided in GNU toolchains to create an executable image with ISR. The overlay feature sets the effective address of each function to the work memory area and the wrapping feature inserts a trampoline before each function. In the trampoline function, we implement an overlay manager that modifies the MPU.

We have implemented two versions of ISR. In the first version each overlay section contains only one function whereas in the second version each section contains multiple functions.

An overlaid function is executed in the following manner. All the access to the work memory is disabled on boot. A call to an overlaid function starts with a call to a trampoline. The trampoline first pushes the return address and currently mapped function load address to a special stack associated with each thread. Then, the currently mapped load address and the load address of the function to be called are compared. If they don't match, all the MPU regions for the work memory are disabled. After that, the trampoline calls the target function.

As the function goes on to execute and accesses an encrypted region, a memory access fault exception is raised. In the exception handler, the corresponding 128B is enabled and decoded into the work memory.

Since the work memory has to be reset between every function call, this makes this approach expensive. To mitigate this, a second optimized version of ISR that puts multiple functions into one overlay based on the execution profile of the first version was conceived.

From execution of the first version, we collect the history of functions decoded into the work memory allowing us to group together those functions with the highest number of occurrences. From this history, we create a weighted graph as in Figure. 6. Each node in the graph represents a decoded function and each edge represents how many times the connected nodes appear together. Consider picking node A, which has the highest occurrence in the graph. We then pick an edge with the largest weight, which is the edge between node A and C in this example. We then check if these two functions fit within the work memory and if so put them in the same group. After that, we pick an edge with the largest weight from this group and continue until all edges are considered. If there is any node that is not part of any group, we pick the one with the highest occurrence

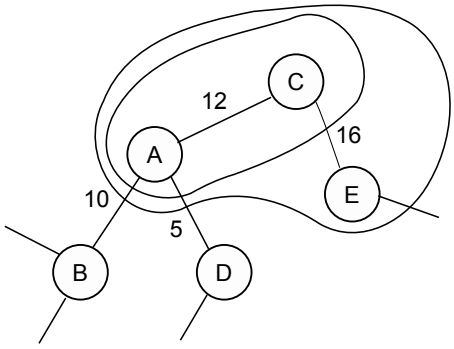


Figure 6: Work Memory Transition Shown as Graph

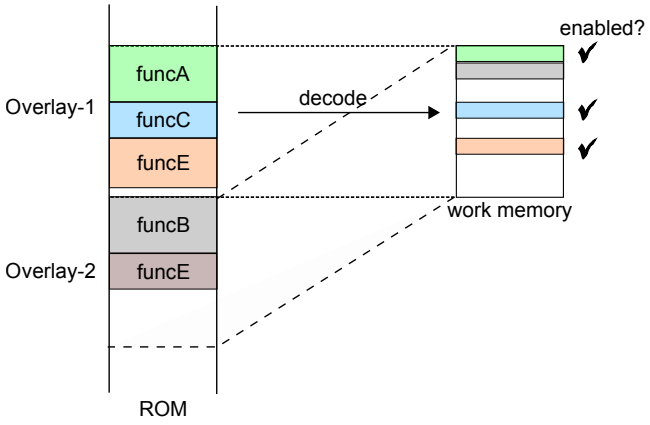


Figure 7: ISR with multiple functions in one overlay section

and repeat the same process.

Note that with our optimized implementation of ISR, 128B blocks in the work memory can contain decoded texts from different overlays as in Figure 7. These blocks are held in the work memory like a cache entry, but any access to them is forbidden by MPU until that overlay section is mapped again. When that region is enabled again, we reuse the cached decoded text.

6. EVALUATION

We analyzed the performance penalty of our proposed rebooting and diversification methods on a CBR600RR engine. Rebooting the ECU has the effect of cutting power temporarily for the time taken to reinitialize the controller. To emulate this behavior, we instrumented a commercial, closed source MOTEC M84 ECU[1] to cut out its output to the engine over a range of engine speeds (hereafter, frequencies) and durations using the powershifting mode to disable injection events for fixed intervals. Using a commercial ECU allows us to analyze performance against a robust implementation which is widely used, expertly tuned implementation on this specific engine, as opposed to our custom implementation and parameters of an open source controller. We then compare these results to the performance penalties realized by our rebooting strategies. To analyze the effect of diversification, we implement our diversification strategies on the opensource rusEFI ECU, and then emulate the engine func-

[1]http://www.motec.com/m84/m84overview/

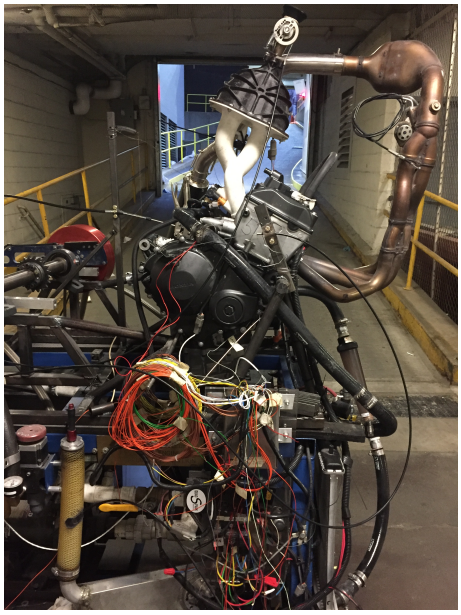


Figure 8: Engine Test Bench with Honda CBR600RR mounted.

tioning at specific speeds while we measure the amount by which scheduled deadlines are missed, and the amount of CPU idle time. We then compare these results with the undiversified version to measure the cost of the diversification.

6.1 Reboot

In this section we first provide the latency of reboots on our RusEFI implementation, and then study the performance costs of rebooting on the MoTEC ECU, and understand the best case performance for rebooting.

Reboot Cost on rusEFI: There are two sources of reboot overheads: the first is simply the cost of rebooting the chip and reinitializing the controller. The second cost is the number of engine cycles taken to measure certain state properties such as engine speed that must be measured over multiple engine periods. If each engine period takes a millisecond (say) then completing these essential tasks will take multiple milliseconds thus limiting the maximum reboot frequency.

The baseline implementation of rusEFI requires 40 ms to restart. We optimized the startup routine by removing trivial functionality like logging, and were able to create a 20 ms reboot. While further optimization may be possible, it will be difficult to realize them without compromising basic security features like wiping the stack region between reboots. The second cost, the cost of multi-period measurements is driven by two main stateful components: these are a) the code to estimate when the trigger is decoded and b) the code that estimates the speed of the engine. Both of these require a full rotation of the engine to warm up their physical state estimation routines. Assuming a nominal engine speed of 4500 RPM (i.e., approx 75 Hz), each engine cycle takes 13 ms. So the state estimation tasks related to YOLO take at another 26 ms (two cycles for estimation). Thus our best case reboot latency is 46 ms. In this time roughly three engine revolutions can complete at 4500 RPM.

Impact of Reboot on Engine Speed/Power: Next, we performed an experiment on a real engine to explore the cost of rebooting as measured by the drop in engine speed, and how it varies with how often the engine is rebooted and the length of each reboot. Running the Honda CBR600RR engine on a test bench (See Figure 8) we captured the engine's rotational speed for different duration and frequencies of ECU reboots.

Figure 9a shows the change in engine speed, and thus stored energy, as the engine power is cut at 1 Hz (i.e., once every second, or once every 75th engine revolution) for durations of 20, 220, and 460 ms. These results show that if the duration and frequency of the reboot is low (20 ms at 1 Hz) then there is no observable loss in engine speed. As the reboot duration increases we see there are significant changes in engine speed.

In addition to engine speed, another metric of interest which is probably more relevant is the engine's rotational kinetic energy. Under normal operation at a constant engine speed, the engine controller maintains the average amount of energy in the system at a relatively constant level. We can measure the degradation in the performance of the ECU as the loss of that average energy as compared to the nominal level. As we increase the frequency of reboots, at some point, the ECU is not able to generate enough energy to overcome friction, and the engine comes to a stop after a number of revolutions, which is ultimately a complete failure of the engine. We refer to the specific engine speed at which this failure occurs as the stalling threshold.

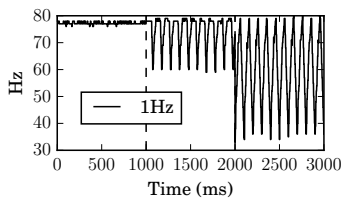
Figure 9b shows the effect of modifying the frequency and duration of the reboot on engine's energy. Each line in the graph represents one frequency of reboot as the duration of the reboot is changed. We plot 1, 2, 4, 8, and 16 Hz reboots (75, 37.5, 18.75, 9.375, 4.687 and engine revolutions respectively at 4500 RPM). The duty cycle of the engine is the fraction of time in which the engine controller is active and able to produce power. As the duty cycle of the engine decreases, the engine loses more energy.

As we reboot more frequently, we observe lower engine speeds without crossing the stalling threshold during operation. The actual stall threshold varies nonlinearly with the frequency and length of the reboots, but we observe that for most of our curves, the engine stalls when it drops below a 50% active duty cycle, with lower engine speeds possible at higher frequencies. This implies that lower latency reboots can be accommodate much higher reboot frequencies without compromising critical functionality, although the performance degradation may be significant.

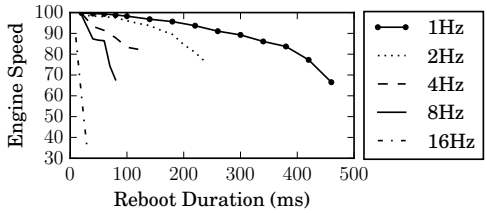
Performance of Optimized Reboots: The purest form of the YOLO design would have us rebooting the engine every time a new sensor input is received, which is once per revolution. However, for our YOLOized ECU, two revolutions are required to derive the minimum state necessary to control the engine correctly. Control output can only be generated during the second of these two revolutions. Assuming the reboot itself takes only as long as a single engine cycle, this results in a 33% duty cycle.

Our results in Figure 9b show that the duty cycle of 33% realized by the idealized YOLO controller will cause the engine to stall, or at least perform very poorly. Even optimizing the reboot time to reduce the amount of time taken to reboot, we will always be below a 50% active duty cycle because of the missed cycle required for state estimation.

Our discussion of these idealized behaviors shows that



(a) Engine Speed @ 1Hz duty cycle (20,220,460ms reboot times)



(b) Engine Speed Reboot Effects

Figure 9: Reboot Evaluation. Measuring the impact of reboot on engine speed/power.

simply rebooting as often on every sensor input is not a viable strategy for maintaining high confidence that the system returns to a well modelled state because the performance penalties are large. This insight demonstrates the necessity of considering less computationally expensive methods such as those described in Section 5.1.4. When no error or inconsistency is detected, these methods suffer essentially no performance penalty. When an inconsistency is detected, our optimized microreboot strategies incur an overhead of one revolution in order to properly recover. For a small reboot period where a single revolution of ignition is missed, there is virtually no distinction between normal engine noise and the effects of rebooting even at 8 Hz, as shown in Figure 9b. So long as an attacker is only able to force the system into an inconsistent state at most eight times a second, we should not observe any performance degradation with the optimized version.

6.2 Diversification

The results of the previous section indicate that microreboot techniques can be used effectively at the granularity of multiple revolutions. To reduce vulnerabilities between reboots we diversify. Diversification when used in conjunction with rebooting allows YOLO to limit the attacker from consistently exploit the system in a predictable manner.

While diversification is a relatively well-studied area, there has been very little work on applying diversification, and measuring the overheads of diversification, on CPS units such as the ECU. As such the goal of our experiments in this section is to determine the limits/applicability of different types of diversification strategies in the context of a YOLOized ECU system. Specifically, we seek to answer the following questions: 1) can diversification be accomplished without harming a CPS operation? In other words can diversification result in missed deadlines? 2) If deadlines are missed how does it impact the overall operation of the system?

Computational Overhead: Here, we present an evaluation of diversification techniques presented in Section 5.2 with respect to the effect that their performance penalties have on our model realtime system, the rusEFI ECU. We applied simulated inputs to the ECU at a range of engine speed inputs. This allowed us to evaluate the overhead of each strategy by comparing the overall computational overhead of the change in latency of hard realtime events as compared to the baseline implementation.

To measure the overall computational overhead, we compared the amount of time that the processor spent idling by examining the cycle counter every time the program entered and left the idle loop. These results are shown in Figure 10a. The idle time performance is shown as a percentage of the total running time. LR^2's performance was found to be on par with the baseline rusEFI primarily due to its efficient load-masking technique. At the other end, we found that both ISR implementations strongly deviated from the baseline. The high overhead is mainly due to the constant copying and decoding of code sections into the work memory, as well as, the overhead of issuing instructions from SRAM. The interesting result is Isomeron and Isomeron with LR^2. The difference in idle time between the two stems from the implementation with respect to code pointer hiding. For Isomeron, a shadow stack is used, while when paired with LR^2 the original stack is used along with XOR encryption resulting in less writes to memory. While we have seen that the different strategies affect the idle time, the context switches to the idle thread remain unchanged since they are not causing processing of events to overlap with each other.

Real-Time Delays: In a CPS, real time events such as sensor inputs are scheduled by configuring a realtime timer to trigger an event handler when it expires. There is some overhead in processing each real time scheduled event due to the scheduler and whatever processing the handler does to calculate the appropriate output for the scheduled event. This overhead is anticipated by the algorithm designer, and must be negligible on the time scale of the expected precision of the CPS. As long as the additional overhead imposed by the diversification method is on the order of these inherent overheads, it can also be considered negligible with respect to the expected behavior of the CPS.

From the perspective of the ECU, the overhead causes errors in the coordination of the event with respect to the physical system. This error may be measured either in terms of time or in terms of the expected angle of the crank shaft. The average overhead for events in the baseline implementation is 80 microseconds, which corresponds to 2.16° at the nominal 75 Hz engine speed described in the previous section. In Figure 10b, we present the computational overhead of each diversification method normalized against this baseline overhead. We see that for LR^2, the computational delay overhead is the lowest at around 13% percent above baseline, corresponding to 2.441°. The hybrid LR^2 and Isomeron have an approximate overhead of 213% (4.601°) while plain Isomeron is 392% (8.467°). Finally, the worst performer is the unoptimized ISR, at around 1476% (31.88°). To put these numbers in context, in commercial systems such as the MoTeC ECU, the real-time events are usually accurate to within $2 - 3°$. Except for the unoptimized ISR most of the diversification strategies fall within the acceptable deviations.

To summarize, what we found was that the hard real-

time deadlines of the ECU can be met despite the increased overhead. While the delay overhead may seem large, due to the time scale at which the physical events occur, the better performing strategies actually have minimal overhead.

7. RELATED WORK

CPS Security: Interest in CPS security is intensifying. Kim et al. [20] reviewed the emerging importance of cyberphysical systems to society at large. Neuman et al. [29] and Cardenas et al. [10] summarized the challenges and criteria to consider in cyber-physical security. Yampolskiy et al. [41] describe a taxonomy of CPS system attacks. Several authors have surveyed existing cyberphysical systems and security paradigms for them, including [19, 26, 35, 21]. Moving target defenses for cyberphysical systems with references to techniques such as diversification and replication were reviewed in detail by Fang et al. [15]. Koscher et al. [22] analyzed modern automobile architectures for security vulnerabilities. Mitchell et al. [28] surveyed many intrusion detection methods for cyberphysical systems. Chow et al. [11] proposed intrusion detection through analyzing data patterns. Burmestera et al. [7] presented a mathematical model for securing distributed cyberphysical systems through byzantine fault tolerance. Fawzi et al. [16] presented a secure controller for distributed linear systems. Pajic et al. [32] presented a model for analyzing controller robustness including timing issues and jitter. Alho et al. [4] presented a service based real-time distributed CPS system with microreboots for teleoperation. Azab et al.[5] described Chameleonsoft, a highly conceptual framework of abstractions for CPS systems that includes a high degree of replication and virtualization, including diversified microservices capable of microreboot. Many of the above defensive works are complementary to proposed work.

Reboot: Resilient operation through recursive microreboot was introduced by Candea et al. [8, 9]. Le et al. [24] explored the additional difficulty of applying this paradigm to low level system software that interacts directly with hardware. Oh et al. [30] introduced a method for detecting errors in computation through replication and handling them by repeating the computation. Several authors ([17, 3, 39] have described detecting and replacing failed or subverted computations through replication and replay.

Diversification and Code Reuse Defenses: Diversification is a popular and expanding security technology that provides resilience against attacks by shifting the attack surface. Larsen et al. [23] and Davi et al. [13] presented a review of the state of the art in protection against code reuse attacks. Davi et al.[12] also presented Isomeron, a quasistatic defense used in this work. Braden et al. [6] presented LR^2, a method for enforcing execute only memory suitable for embedded devices. Pappas et al. [34] introduced ISR, a code obfuscation method that hides usable ROP widgets.

While some of these works have analyzed security approaches for CPS systems in general, to the best of our knowledge none have directly approached the problems of implementation of these ideas in the context of low-level controllers with high computational demands. Our work aims to address this gap.

8. CONCLUSION

We present a new security paradigm called YOLO, that leverages unique properties of cyber-physical systems such as inertia and control algorithm resilience by combining diversification with micro-rebooting. The paradigm aims to emulate the inherent security and robustness properties of a stateless, idempotent system even though cyber-physical systems are generally stateful and not fully observable. We compensate for these violations of the ideal situation by carefully considering how system state should be treated to be verified or regenerated on demand. We demonstrate YOLO on an Engine Control Unit. We detail the challenges of dealing with state used by control algorithms to estimate the physical state of the system and implement solutions to address them through a combination of techniques that support the complex, nonlinear state estimation required for correct engine operation. Our experiments demonstrate that YOLO is possible on a real world CPS. We show that the overheads imposed by the instrumentation necessary for YOLO are tolerable by a real engine.

9. REFERENCES

[1] Design Flaws Expose Drones to Hacker Attacks: Researcher | SecurityWeek.Com.

[2] The Real Story of Stuxnet - IEEE Spectrum.

[3] A. Agbaria and R. Friedman. Overcoming Byzantine Failures Using Checkpointing. *System*, pages 1–11.

[4] P. Alho and J. Mattila. Service-oriented approach to fault tolerance in CPSs. *Journal of Systems and Software*, 105:1–17, jul 2015.

[5] M. Azab, V. Tech, R. Hassan, C. Science, M. Eltoweissy, and P. Northwest. ChameleonS oft : A Moving Target Defense System. *Components*, pages 241–250, 2011.

[6] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, A.-r. Sadeghi, and U. Darmstadt. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*, number February, pages 21–24, 2016.

[7] M. Burmester, E. Magkos, and V. Chrissikopoulos. Modeling security in cyber–physical systems. *International Journal of Critical Infrastructure Protection*, 5(3):118–126, 2012.

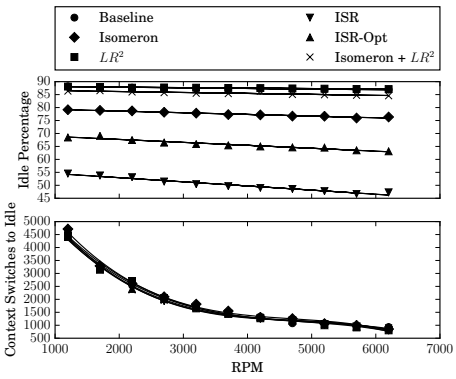
[8] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Performance Evaluation*, 56(1-4):213–248, mar 2004.

[9] G. Candea and A. Fox. Crash-Only Software. *Hot Topics in Operating Systems*, IX(May):19–24, 2003.

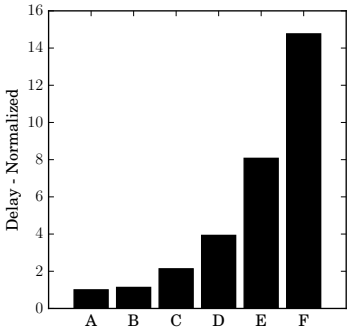
[10] A. Cardenas, S. Amin, B. Sinopoli, A. Giani, A. Perrig, and S. Sastry. Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, page 5, 2009.

[11] R. Chow, E. Uzun, A. Cárdenas, Z. Song, and S. Lee. Enhancing Cyber-Physical Security through Data Patterns. *Workshop on Foundations of Dependable and Secure Cyber-Physical Systems*, page 25, 2011.

[12] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings 2015 Network and Distributed System Security Symposium*, Reston, VA, 2015. Internet Society.



(a) Idle Performance



(b) Computational Delay – **A**:Baseline **B**:LR^2 **C**:Isomeron + LR^2 **D**:Isomeron **E**: ISR-Opt **F**:ISR

Figure 10: Diversification Evaluation. Measuring the effects of diversification strategies on ECU delays.

[13] L. Davi and A.-R. Sadeghi. *Building Secure Defenses Against Code-Reuse Attacks*. SpringerBriefs in Computer Science. Springer International Publishing, Cham, 2015.

[14] Y. Ding. Recovery-Oriented Computing : Main Techniques of Building Multitier Dependability. 1, 2007.

[15] S.-w. Fang and A. Portante. Moving Target Defense Mechanisms in Cyber-Physical Systems. In *Securing Cyber-Physical Systems*, chapter 3, pages 63–90.

[16] H. Fawzi, P. Tabuada, and S. Diggavi. Secure Estimation and Control for Cyber-Physical Systems Under Adversarial Attacks. *IEEE Transactions on Automatic Control*, 59(6):1454–1467, jun 2014.

[17] L. Hatton. N-version design versus one good version. *IEEE Software*, 14(6):71–76, 1997.

[18] R. Ivanov, M. Pajic, and I. Lee. Attack-Resilient Sensor Fusion for Safety-Critical Cyber-Physical Systems. *ACM Transactions on Embedded Computing Systems*, 15(1):1–24, 2016.

[19] S. K. Khaitan and J. D. McCalley. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal*, 9(2):350–365, 2015.

[20] K.-D. Kim and P. R. Kumar. Cyber-Physical Systems: A Perspective at the Centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue):1287–1308, 2012.

[21] R. a. Kisner, W. W. Manges, L. P. MacIntyre, J. J. Nutaro, J. K. Munro, P. D. Ewing, M. Howlader, P. T. Kuruganti, R. M. Wallace, and M. M. Olama. *Cybersecurity through Real-Time Distributed Control Systems*. Number February. 2010.

[22] K. Koscher. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2012.

[23] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. *2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[24] M. Le and Y. Tamir. Applying microreboot to system software. *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability, SERE 2012*, pages 11–20, 2012.

[25] R. M. Lee, M. J. Assante, and T. Conway. German Steel Mill Cyber Attack. pages 1–15, 2014.

[26] T. Lu, J. Zhao, L. Zhao, Y. Li, and X. Zhang. Towards a framework for assuring cyber physical system security. *International Journal of Security and its Applications*, 9(3):25–40, 2015.

[27] M. Mery. Explosive engine, July 23 1895. US Patent 543,157.

[28] R. Mitchell and I.-R. Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Comput. Surv.*, 46(4):55:1–55:29, Mar. 2014.

[29] C. Neuman. Challenges in Security for Cyber-Physical Systems. *Workshop on Future Directions in Cyber-physical Systems Security*, pages 1–4, 2009.

[30] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[31] D. Oppenheimer, D. Patterson, and A. Ganapathi. Why do Internet Services fail, and what can be done about it? *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems-Volume 4*, pages 1–1, 2003.

[32] M. Pajic, J. Weimer, N. Bezzo, and P. Tabuada. Robustness of attack-resilient state estimators. pages 163–174, April 2014.

[33] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 981–992. ACM, 2013.

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis. Practical software diversification using in-place code randomization. In *Moving Target Defense II*, pages 175–202. Springer, 2013.

[35] A.-S. K. Pathan. *Securing Cyber-physical Systems*. CRC Press, 2015.

[36] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48, New York, NY, USA, 2010. ACM.

[37] G. Portokalidis and A. D. Keromytis. Global isr: Toward a comprehensive defense against unauthorized code execution. In *Moving Target Defense*, pages 49–76. Springer, 2011.

[38] D. P. Shepard, J. A. Bhatti, T. E. Humphreys, and A. A. Fansler. Evaluation of smart grid and civilian uav vulnerability to gps spoofing attacks. In *Proceedings of the ION GNSS Meeting*, volume 3, 2012.

[39] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, page 7, 2004.

[40] Wikipedia. Powershifting, 2016. [Online; accessed 11-May-2016].

[41] M. Yampolskiy, P. Horvath, X. D. Koutsoukos, Y. Xue, and J. Sztipanovits. Taxonomy for Description of Cross-Domain Attacks on CPS.