Experimentation and Implementation of BFT++ Cyber-attack Resilience Mechanism for Cyber Physical Systems

DAVID R. KEPPLER, MITRE, USA
M. FARAZ KARIM, CSAFA Labs, SCP, Georgia Institute of Technology, USA
MATTHEW S. MICKELSON, MITRE, USA
J. SUKARNO MERTOGUNO, CSAFA Labs, SCP, Georgia Institute of Technology, USA

Cyber-physical systems (CPS) are used in various safety-critical domains such as robotics, industrial manufacturing systems, and power systems. Faults and cyber attacks have been shown to cause safety violations, which can damage the system and endanger human lives. Traditional resiliency techniques fall short of protecting against cyber threats. In this paper, we show how to extend resiliency to cyber resiliency for CPS using a specific combination of diversification, redundancy, and the physical inertia of the system.

Additional Key Words and Phrases: cyber-physical systems (CPS), safety-critical systems, cyber resilience, fault tolerance, inertia

1 INTRODUCTION

Cyber-physical systems (CPS) are required to satisfy safety constraints in various application domains such as robotics, industrial manufacturing systems, and power systems. Once isolated system of computer controlled machinery are now more exposed to the external world than ever. This not only leaves the traditional threat of a physical component failing but also of a cyber threat of a remote disruption in the system. Our goal with Byzantine Fault Tolerance (BFT++) [16] is to force any cyber disruptions to be short-lived. This brittleness of the system, along with timely and stateful recovery, within tolerance of the physical system's inertia, presents a robust cyber attack tolerance mechanism.

Two limitations of modern CPS make securing them a challenge. First, many CPS are built on legacy systems. Second, no system is devoid of bugs and exploitable vulnerabilities. Our approach addresses both these points, allows for BFT++ to be retrofit onto legacy systems. The flexibility provided by our technique sets it apart from other modern CPS resiliency techniques which are more focused on anomaly detection, access control (e.g. encryption), formal methods, and so on.

2 BACKGROUND

2.1 Contemporary CPS Defenses

Modern-day cybersecurity protections can be categorized into three key areas. *Prevention*, *Detection*, and *Response* [7]:

Authors' addresses: David R. Keppler, dkeppler@mitre.org, MITRE, McLean, Virginia, USA, 22102; M. Faraz Karim, mkarim40@gatech.edu, CSAFA Labs, SCP, Georgia Institute of Technology, Atlanta, Georgia, USA, 30332; Matthew S. Mickelson, mmickelson@mitre.org, MITRE, McLean, Virginia, USA, 22102; J. Sukarno Mertoguno, karno@gatech.edu, CSAFA Labs, SCP, Georgia Institute of Technology, Atlanta, Georgia, USA, 30332.

ACM 2378-962X/2024/1-ART
https://doi.org/10.1145/3639570

- *Prevention* seeks to stop attackers from gaining system access via measures such as access controls and authentication. Our approach, BFT++, aims to provide resiliency and relies less on preventing access.
- *Detection* seeks to identify anomalous behavior in a system in case prevention measures fail. Our approach does not rely on detecting malicious input, BFT++ works by translating attempts to seize control into a system crash which subsequently triggers the resiliency mechanism.
- *Response* seeks to mitigate and remediate the attack after detection. Our approach seeks to continue physical operation despite faults introduced by any disruption; cyber or otherwise. Further, our approach does not require any diagnosis of the root cause of the fault.

Similarly, many modern-day CPS cyber protections can be classified as follows. These Approaches are also represented in *Table 1* [27] along with further sub-classifications.

- *Domain-Specific Analysis* solutions aim to systematize attacks and defenses in specific application domains of CPS, e.g., power grids[14] [15], medical devices [23] [4], industrial control systems (ICS) [13] [24], etc. BFT++, is domain agnostic as we are trying to protect the physical processes instead of the cyber.
- *Algorithm-Level Analysis* solutions focus on exploring different existing algorithms for intrusion detection [10] [17] and anomaly identification[8] in CPSs. Our solution actually does not detect the malicious presence in the system and just waits for a system crash to occur due to malicious input.
- *Sensing Pipeline Analysis* solutions focus on how maliciously crafted inputs affect sensors in the CPS; which may lead to out-of-band injection vulnerabilities [5][6] or transduction attacks [26]. Our work focuses on maintaining operation of the physical process; we are not concerned with the vulnerabilities of the physical system and can continue to provide resilience even in the presence of such a vulnerable system.

Generally, many CPS defenses deployed in operations are extensions of traditional network and host protections. As such, they tend to focus on the human-machine interface (HMI), monitoring of network and the security information, or the event management system (SIEM) levels. However, this still leaves the lower levels of the system vulnerable.

Protection of low-level networks includes protecting the interconnect and computation or logic of the controllers. Cryptographic protections also provide a way to disrupt potentially rogue modules from snooping on the bus. Though it can be effective against some threats, encryption and network monitoring at this level is often impractically expensive and non-scalable, especially when retrofitted to existing systems. Compromises at the controller level also render encryption irrelevant (and also protect attack traffic).

Protecting the controllers themselves involves fault avoidance, fault tolerance, and formal methods that reason over certain properties of a specification. Considering the practicality of both models, formal methods are an excellent approach for achieving CPS security but would imply the complete redevelopment of the systems from scratch. This is expensive for legacy systems, which are prevalent in this setting. We explore a new approach to fault tolerance for CPS called BFT++. In our approach, components are engineered to be brittle, which forces cyber-induced disruptions to be within the tolerance of the physical system's inertia.

2.2 Scope and Novelty of Our Approach

Our goal is resilience of the physical process. As such, we make no assumption that a system is devoid of bugs. Rather, we seek to enable a CPS to continue physical operation regardless of cyber-induced faults. Our goal is to address a well-resourced adversary whose goal is to subvert control of the physical process. Generally, there are two ways to subvert or negatively affect the behavior of a controller: (a) manipulate its inputs to cause improper control output or (b) alter the controller's programming or configuration to hijack its execution. Note that cyber attacks on confidentiality can gather intelligence, and help plan for an attack, but by themselves do not affect or subvert the physical process.

Table 1. Categorical summary of different approaches in CPS security based on surveys.[27]

Approach	Specific Domain	Reference
Domain-specific analysis	Power grid	[14],[15],[12],[19]
	Medical device	[23],[4]
	Industrial control system	[13],[24]
	Drone	[1]
Algorithm-level analysis	Secure estimation & control	[25]
	Intrusion detection	[10],[17]
	Anomaly identification	[8]
Sensing pipeline analysis	Out-of-band injection	[5],[6]
	Transduction attack	[26]

Our approach does not seek to address "malicious insider" attacks focused on co-opting legitimate control or configuration tooling of the system, e.g., engineering workstations or human-machine interfaces (HMIs). Nor do we address network integrity threats such as adversary-in-the-middle attacks on sensor or HMI values. These risks are better addressed through other means, such as multi-factor authentication, workstation endpoint monitoring, and network encryption.

Our approach *does* seek to address attacks that alter a controller's behavior via directly exploiting a vulnerability in its code, e.g. code injection via a memory safety vulnerability. Our methods empirically show resilience of the physical operation against this particular class of hijacking. For example, we can protect a system's physical operation from an exploited controller able to send or inject inputs.

2.3 Unique Advantages of Securing CPS

The physical aspects of CPS enable certain advantages not available in IT systems. Specifically, the physical processes allow for a certain degree of predictability (following the rules of physics) in the behaviors of the system. The three key characteristics of CPS that we focus on are periodicity, physical inertia, and the commonality of legacy fault tolerance techniques.

Periodicity: The cyber subsystem directly interacts with the physical subsystem in a continuous and repeating loop. Throughout its execution, the controller reads values from sensors, performs some data processing, and produces actuator values. For the commonly used class of industrial Programmable Logic Controllers (PLCs), this loop is called the scan cycle. This periodicity allows an opportunity to recover from a fault affecting a single cycle in a predictable manner; safely regaining control of the physical process in later cycles.

Inertia: Any physical subsystem of a CPS must obey the laws of physics and physical systems have inertia. The scan cycle of a controller is typically engineered to be fast enough (e.g., 1 Hz to 1 kHz) such that an issue in a small number of cycles will be dampened by the existing inertia. Importantly, this scan cycle is orders of magnitude slower than processor cycles. Taken together, this means the physical process cannot change by large amounts in a small amount of time; allowing us small time frames to recover the physical process from any faults.

Existing Fault Tolerance: We intend to build upon existing fault tolerance design elements when present in the legacy system. Most safety-critical systems will utilize some type of redundant architecture to deal with faults. Approaches could include: hot backups; dual, triple, or quad-redundant architectures; or byzantine fault tolerance where assumptions about the error conditions are minimal, and random faulty replicas may behave arbitrarily.

3 CYBER RESILIENCE STRATEGY

Cyberattacks, however, may drive traditional fault-tolerant systems into common-mode failures. Worse, if the attacker is successful in compromising a component, there is no obvious fault signal to detect, and the controllers continue to actuate the system while under the control of an adversary. Attempts to deal with common-mode failures have been made through diversification, but the type of diversification used must be matched to the class of causes of common-mode failures that the system owner wants to mitigate. We will show that resiliency to cyberattacks can be achieved using diversification that can disrupt the attacker's exploitation attempts.

3.1 Exploit Disruption

The process of a cyber exploit involves two virtual stages: first, exploiting a flaw/vulnerability in the program's code to alter its intended execution path, and second, taking control of the system to execute the attacker's commands. This is analogous to a fumble in football, where an opposing team must not just cause a fault, but recover the ball to gain possession, as shown in Figure 1.

The first step in exploit disruption is to prevent the multiple redundant controllers from simultaneous compromise. To do this, we use artificial software diversification. By shuffling program memory in a way that makes it difficult for attackers to find the proper target code address, this diversity can disrupt the exploitation process. When used in combination with traditional fault tolerant architecture, this helps ensure a potential exploit has different effects on diversified controllers. Diverse replicas will have different memory layouts, making it almost impossible for attackers to inject malicious code that works across all replicas simultaneously. Behavior of an attack across replicas will necessarily vary, creating an opportunity for detection.

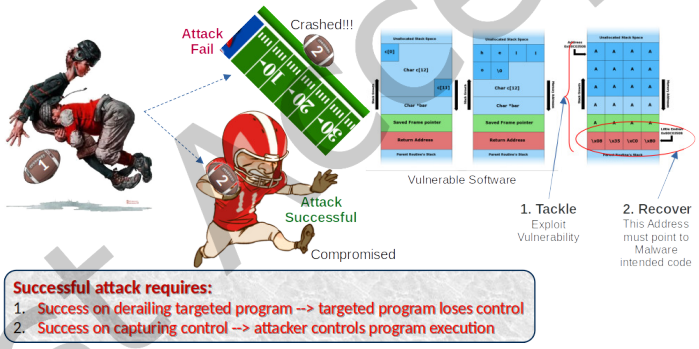


Fig. 1. Two Virtual Stages of Cyber Exploit.

3.2 Automating Recovery

The next crucial step is enabling the system to recover. Diversity within the system can make it more fragile, so fast-acting and automated recovery must be employed to counterbalance this. Without recovery, the attacker could maintain control of one (compromised) replica and leave the others crashed — a clearly unacceptable state.

Crash detection is the first part of the recovery process. Ideally, we want to detect a potential compromise (via a crash of one of the diversified replicas) as soon as possible. In our case, a replica failing to produce timely output by the end of the epoch is considered to have crashed. This serves as a canary that there has been a compromise.

Next, a small message queue is employed in front of one of the replicas (henceforth referred to as the "protected" or "delayed" controller). This is key, because when a potential compromise is detected (via the crash detection), the message(s) triggering said crash are trapped in the queue before reaching the protected replica. Upon crash

detection, this queue can be flushed removing the offending messages. While this introduces a small delay to the protected controller, we shall see in later sections how the physical inertia of the system allows us to absorb this without impact to the real-time operation.

Finally, recovery begins, and the state of the replicas are restored. Restoring from a checkpoint is possible, but requires enough resources to handle the overhead of saving checkpoints as well as a way to deal with staleness of state upon a restore. Instead, the strategy advocates designating one or more replicas as backups and time-delaying them so they process inputs one or more cycles behind the primaries.

3.3 Implementation Theory

The BFT++ architecture assumes three redundant diversified controllers operating in a traditional fault-tolerant architecture. The artificial diversity makes it difficult for a cyber attacker to compromise all controllers with the same malicious input. Although an exploit may be successful against one replica, it will cause the diversified replicas to crash. Next, it incorporates delayed input sharing (e.g., a first-in-first-out (FIFO) message queue) to trap bad messages before reaching a "protected controller". This introduces a delay to the protected controller, but ensure the system can continue operation and be reconstituted after the cyber exploit. For a given BFT++ implementation, the recovery timing of the system is governed by several timing parameters, such as T_{crash}, T_{sc}, D, T_d, and T_r. T_{sc}, T_r, T_{sc} are system parameters, and D needs to be appropriately set for automated recovery to be possible. Section 4 describes our specific experimental architecture, and includes several diagrams of this conceptual setup.

The two critical points that determine the system's recovery timing are the brittleness of the controllers (e.g., how quickly failures occur) and how quickly the system can restore a controller with state. The physical subsystems with higher inertia are generally more tolerant of losing control signals for a short time. In general, the following relationship between these parameters must hold for BFT++ to be applicable to a legacy system:

$$T_{crash} \leq D * T_{sc} \leq T_d - T_r$$

Parameters	**Definitions**
T_{crash}	Time between attack and crash
T_{sc}	The scan cycle period
D	FIFO queue length (number of slots)
T_d	Maximum control loss tolerable by physical systems
T_r	Recovery latency

4 IMPLEMENTATION

4.1 Demo Application

For the experimental implementation, the team utilized a Fischer Technik Vacuum Gripper Robot (VGR) as the physical system under the control of Siemens S7-1500/1200 family programmable logic controllers (PLCs). The VGR consists of a gripper manipulator attached to a 3-axis arm that can position the manipulator through rotational, vertical, and horizontal movement. A vacuum pump powers the manipulator head which can then grip and hold small objects (Figure 2a). The VGR is one of four components in a larger Factory Simulation that represent the elements of a complete factory production line (Figure 2b).

The various sensors and actuators found in the Factory Simulation (e.g., motors, encoders, limit switches, solenoids, etc.) are designed to directly interface with industrial controllers such as the Siemens PLCs we selected, allowing for convenient realistic experimentation. Similar controllers from other PLC manufacturers could be utilized instead of Siemens without significant changes to the BFT++ architecture. The properties of the robot arm

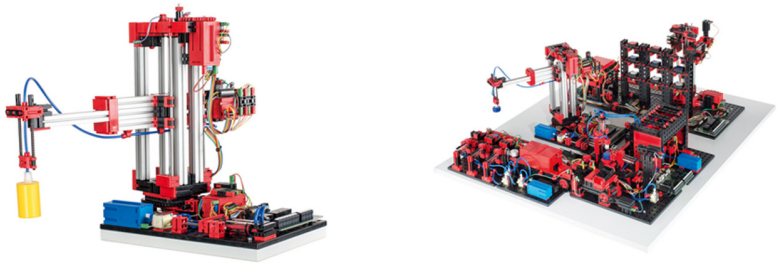


Fig. 2. Experiment Setup. (a) Vacuum Gripper Robot, (b) Factory Simulation.

make it a broadly representative example of a physical system with an element of inertia, with readily apparent similarities across many cyber physical domains.

To implement the VGR's range of actions, it contains several sensors and actuators for moving the arm, measuring its position, and activating the vacuum gripper manipulator. In total, the device has eight 24V digital inputs:

- Forward and reverse controls for the motors on each of the three arm axes: clockwise and counterclockwise rotation; horizontal extension and retraction; and vertical raising and lowering.
- Activation of an air compressor to drive the pneumatic vacuum system.
- Solenoid valve actuation signal that releases compressed air into the cylinder arrangement creating a suction effect at the manipulator head.

Several output signals provide information about the positioning of the arm:

- Encoder signals from the motor on each arm axis that work together to emit pulses allowing a control system to count that motor's revolutions in each direction.
- A limit switch on each axis that emits a signal when the arm hits the "home" position of the axis representing zero in the coordinate system for that axis.

These signals are connected to a Siemens SIMATIC ET 200SP Distributed Input / Output (I/O) module containing several digital I/O blocks and high-speed counter modules. The ET 200SP is connected to a Siemens SIMATIC S7-1500 PLC via the PROFINET protocol via Ethernet (Figure 3a).

The Siemens S7 logic controller implements a program that drives the functionality of the robot arm. It is primarily driven by commands input into the controller's memory over the network from a higher-level HMI or SCADA system. These commands consist of a string of movement and action directives (e.g, move to location A, activate gripper, move to B, deactivate gripper, and so on). Commands are executed by a state machine loop that:

(1) If not busy with a currently executing command, wait for a start signal input then copy the input command into working memory.
(2) "Homes" or "zeroizes" the arm axes to ensure calibration of the movement coordinate space. Specifically, run the motors in the negative direction until the limit switch on each axis is hit, then marking that at the "0" location. This is done at the beginning of every command to avoid accummulation of position error.
(3) For each entry in the working command string:
  (a) Look up the arm axis coordinates corresponding to the given location label in a table.
  (b) Drive the motors in the needed direction to reach the desired location.
  (c) Perform an optional action while at that target location (gripper on/off, pause movement).

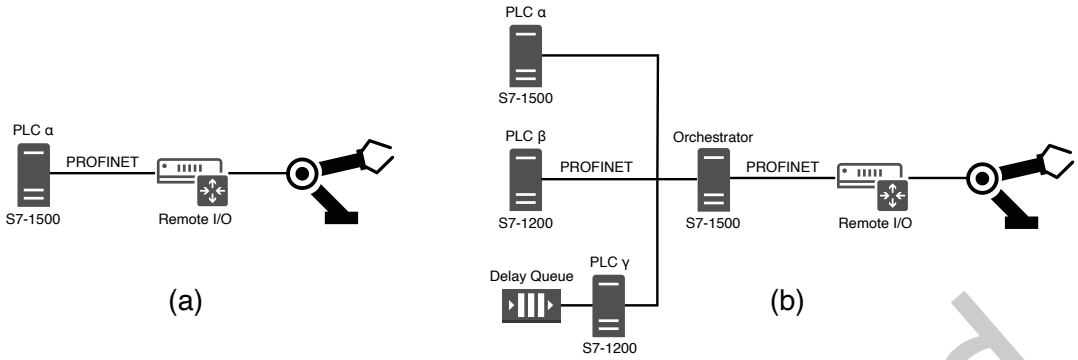


Fig. 3. System Architecture. (a) Pre-BFT++, (b) Post-BFT++.

4.2 Design Goals

Enhancing the vacuum gripper scenario with BFT++ has several requirements:

(1) **Safety.** The implementation must maintain the safety of the system. While the worst thing that can happen in the factory simulation is stripping a gear or breaking a plastic piece, losing control of a real-world piece of industrial equipment can easily lead to equipment damage, injury, or even loss of life. Application of BFT++ to the control system must result in a safe state for the physical system, despite the occurrence of cyberattacks. The vacuum gripper actuator should keep hold of a puck throughout the duration of an attack without dropping it unexpectedly, and the arm's movements should stay within safe bounds without colliding with other pieces of the mechanism or the arm's end stops.
(2) **Availability.** Application must maintain the availability of the system. The process of moving and placing pucks should continue with minimal interruption throughout the duration of the attack. The system should be able to detect the attack, fail over to a protected PLC, reset the affected components, and return to normal operation automatically.
(3) **Discreteness.** The retrofit should be minimally invasive and not increase the attack surface of the system. Ideally, the resulting modified control system would appear to the outside world as if it were a single system with an equally small network footprint. That is, an ideal BFT++ "black box" implementation would look and act indistinguishably from the single PLC it is replacing, and not expose any of its internal implementation to outside attackers.

To meet these requirements, we extended the system architecture shown in Figure 3a with additional components shown in Figure 3b. The first change is the addition of two more logic PLCs (β and γ) to the original S7-1500 (PLC α). PLC β, a S7-1200, serves as one half of a diversified pair with PLC α, with part of the diversification provided by using different hardware models, with diversified firmware, for each device. PLC γ, another S7-1200, acts as the backup to take control of the process when α or β fail. A network delay queue device, a standard Linux computer with multiple Ethernet interfaces, is placed between the network switch and PLC γ and acts as a network gateway while adding a time delay to the message transiting it to PLC γ. The gateway / delay queue machine also acts as reverse proxy exposing a unified Open Platform Communications (OPC) Unified Architecture (UA) protocol interface for the VGR apparatus to outside world as if the trio of backend logic controllers were a single device. Finally, an Orchestrator PLC is added in between the trio of logic PLCs and the ET200 remote I/O. This device has two roles. It acts as an I/O multiplexer, determining which control logic PLC's outputs are forwarded to the physical apparatus and broadcasting the returning input signals to all three logic PLCs. The

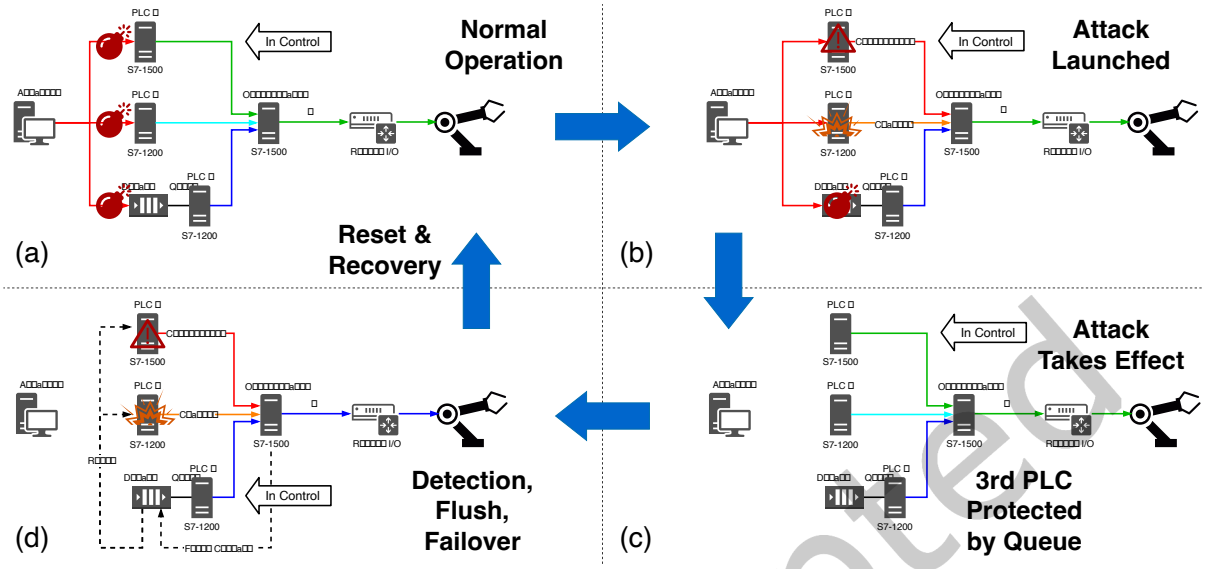


Fig. 4. BFT++ Operational Sequence: (a) Normal Operation, (b) Attack Launched, (c) Effect of Attack, (d) Detection, Failover, and Recovery. Note the arrow indicating which logic PLC is in control of the physical process at each stage.

Orchestrator is also responsible for detecting an attack on the logic PLCs, switching control to and from the delayed backup, and orchestrating the reset and recovery of the logic PLCs.

4.3 Operating Flow and Recovery Concept

In the face of attacks, the BFT++-protected system cycles through several states to blunt the attack, maintain safe control of the physical process while degraded, and then recover the control system to a safe state. This cycle is depicted in Figure 4.

During normal operation without an attack present, all three logic PLCs execute the control logic; see Figure 4a. Input sensor readings from the physical system are received by the Orchestrator I/O multiplexer from the remote I/O and redistributed to each logic PLC. The logic PLCs process these inputs and yield output signals destined for the physical system. On each iteration of the control loop, the logic PLCs copy their outputs to the Orchestrator, which then decides which set to forward on to the remote I/O. During the normal operating state, the Orchestrator selects PLC α's outputs and the others are dropped.

Communications between the outside world and the control system are governed by the protocol gateway / delay queue mechanism. This limits external communications to a single protocol (in this case OPC UA) and IP address. The gateway translates OPC UA inputs to equivalent reads and writes to the logic PLCs via the Siemens S7 management protocol and duplicates these across all three logic PLCs. Messages bound for PLC γ must first traverse the delay queue. See Figure 4b.

Next, the non-delayed logic PLCs receive and begin to process the message and trigger a synthetic vulnerability we added to the system. Unlike the non-BFT++ case, the logic PLCs are diversified so that the attack does not behave the same way on each device. We combine software and hardware diversity to achieve this. In the best case for the adversary, the attack will succeed in compromising one of PLCs α and β (Figure 4c). The diversification applied also has the property of causing the PLC to fail fast and stop executing its control loop when the attack is not specific to that PLC. Therefore, as a result of the attack, PLC α will be in a compromised state, while PLC β

will have effectively crashed. The attack message will still be held in the delay queue preceding PLC γ. At this stage, where the attack has just begun to take effect, the Orchestrator PLC is not yet aware of the compromise and crash of β and continues to pass α's outputs to the physical system. Now the physical system's inertia comes into play to protect the system. The execution speed of the PLCs, as compared to the limits imposed by the mechanics and physics of the system, create a window of time where malicious control outputs are reaching the physical apparatus, but they have not yet had enough time to have a noticeable effect and overcome the system's inertia. The attack has commanded the vacuum gripper to release its hold; however, it takes time for the gripper to lose enough suction to be overcome by the force of gravity, causing the puck to fall irretrievably far away from the gripper head where it cannot be caught again.

Time passes, and the Orchestrator becomes aware that one of the logic PLCs has stopped forwarding its outputs. The failover and recovery process can now begin. Immediately upon attack detection, the stream of control logic outputs being forwarded to the remote I/O and physical system is switched from PLC α to the "not yet compromised" PLC γ. Assuming the operator has tuned the attack detection timings and delay queue size correctly, the malicious message will still be in the queue and not yet have reached γ. Next, the Orchestrator commands the delay queue to be flushed, eliminating the attack data, and maintaining PLC γ in an uncompromised fully functional state. With PLC γ safe, and in control of the physical system, its control logic will begin to return the physical system to within its safe margins. In this case, assuming the detection system timings are tuned properly in comparison to the physical inertia of the system, the gripper is reenabled so that it catches the payload before it has fallen significantly. See Figure 4d.

Despite having protected PLC γ and prevented the attack from having a persistent malicious effect on the physical system, the system is operating at reduced capacity. With only a single operating logic PLC active, the system is vulnerable to re-attack. Without the detection capability yielded by the diverse pair of α and β, a subsequent attack may pass through the delay queue and compromise or crash PLC γ. The Orchestrator must reset PLC α and β, clear the effects of the attack, and bring them back into service. Once the Orchestrator has established that α and β are functional and responsive, the system transitions to the normal operating state and begins forwarding PLC α's outputs to the remote I/O and physical system once again. See Figure 4a.

4.4 Component Implementations

The following sections describe the implementation details of the added components described above, as well as the modifications required to the pre-existing control logic program as it is duplicated across multiple logic PLCs.

4.4.1 Orchestrator. The Orchestrator must adhere to several requirements to support the BFT++ scheme. As a multiplexer for I/O signals to and from the hardware, the Orchestrator must:

- Broadcast all sensor readings received from the remote I/O module to all PLCs.
- Select a single PLC to be in control of the physical system at a given moment.
- Transmit outputs from the previous PLC to the remote I/O module.

Implementing the BFT++ scheme adds the additional requirements to:

- Detect when the logic PLCs are attacked.
- Flush the delay queue.
- Coordinate failover to the protected PLC.
- Recover the logic PLCs to a safe and operational state.

Implementing the Orchestrator starts with the need to multiplex and proxy the inputs and outputs between the logic PLCs and the hardware. In the case of our setup, this pathway already uses a network abstraction for remote I/O (the ET 200SP), which simplifies the problem. In many cases, the PLC is directly attached to the digital and analog I/O of the process hardware. In either case, most PLCs use some form of memory mapped I/O,

which helps create an abstraction layer. That is, the sending and receiving of signals to the hardware are mapped to reads and writes to specific registers or addresses in the device's memory. The next step is to replicate this mapping on the Orchestrator, so it mirrors the process hardware of the non-BFT++ logic PLC. At this point, the Orchestrator can communicate with the process hardware the same way as the original.

Next, we create a network communications path between the logic PLCs and Orchestrator. Many protocols will work, but in the Siemens ecosystem, the PROFINET protocol is well integrated into the programming model for the devices, and abstracts away the complexity of managing a network connection and transferring I/O between devices. Ideally, the communications and multiplexing process proceeds in this repeating cycle:

(1) The Logic PLC program writes to a particular set of I/O locations. In the non-BFT++ case, these would be memory mapped I/O locations corresponding to the process hardware. In BFT++, they redirect to a special set of data structures reserved for use by the BFT++ scheme.
(2) A BFT++ component added to the Logic PLCs (e.g., a software or hardware shim[1]) takes the data written to those locations and communicates them over the network to the Orchestrator. This can be as simple as copying the data to a different set of memory mapped I/O addresses. This is replicated on each of the three Logic PLCs.
(3) The Orchestrator receives data from each Logic PLC and stores it temporarily in memory structures that mirror the original memory mapped I/O.
(4) Based on the crash detection and recovery logic, the Orchestrator then decides which data from the Logic PLCs to write out to the actual process hardware.

Then, in the opposite direction:

(5) The PLC hardware receives sensor readings and delivers them into memory mapped I/O on the Orchestrator the same way as implemented in the non-BFT++ implementation.
(6) The Orchestrator copies these sensor inputs over the network to the logic PLCs where they are received by the BFT++ shim on the Logic PLC.
(7) The shim then places the input data into memory locations that the (otherwise unmodified) control logic program can process to compute the next set of outputs to begin the cycle anew.

We implemented this schema in the VGR testbed as follows. First, we examined the logic PLC code to identify the memory locations corresponding to the process hardware inputs and outputs. Using the Siemens TIA Portal development environment, we back-traced from the hardware I/O modules to find their corresponding memory address in the Input and Output address spaces, and finally identified the named variable tags associated with those addresses.

PROFINET was the ideal network protocol to create the data transfer shim. The team configured each logic PLC as a PROFINET "I/O device" with the Orchestrator as the PROFINET "controller." Input and Output memory address ranges were selected on each side of the connection. From there, the PLC firmware's PROFINET mechanism automatically copies data from the "output" address on one side of the connection to "input" addresses on the other and vice versa. This mechanism takes care of steps 1-3 and 6-7. PROFINET is designed for reliable, low-latency communication between PLCs, so it also meets the other requirements for a network medium for the Orchestrator. As the Orchestrator is now the node that communicates (via PROFINET) with the Remote I/O component, the team simply replicated the Input and Output memory mapped I/O address arrangement from the unmodified Logic PLC program on the Orchestrator. The remaining steps 4 and 5 are a matter of copying variables from one set of addresses to another based on the multiplexing decision signal.

To allow the Orchestrator to detect when a logic PLC has crashed — its signal that an attack may be underway and the queue flush and recovery process should be initiated — we add code to each logic PLC implementing a

[1]In this context, a shim refers to an intermediary allowing for the capture and instrumentation of I/O

liveness counter. This counter value is transmitted to the Orchestrator with the rest of the logic PLC's output data. The Orchestrator monitors these liveness inputs. If a liveness counter does not update within a certain time, the Orchestrator determines a crashed state.

4.4.2 Attack Detection. The resiliency properties BFT++ provides depend on quickly detecting a fault and flushing the corresponding message from the delay queue before it can reach PLC γ. This is accomplished through diversifying the implementations of PLCs α and β in a way that prevents simultaneous failure by the same malicious payload. Furthermore, an unsuccessful attack on a PLC should result in a crash as quickly as possible after the initial attack attempt. Any delayed response would allow an adversary time to establish a foothold before reset and recovery.

Controller diversity may be accomplished in multiple ways. One option is to employ different hardware controllers, e.g., different vendors or models as we did. Another is to employ software diversification techniques on the controller firmware. Either of these techniques would make it more costly for an adversary, and extremely unlikely, to craft a single attack payload that simultaneously compromises both platforms at the operating system and firmware levels. This provides basic protection against attacks on operating system services but does not provide the desired "fail fast" behavior. This residual issue will be resolved through other aspects of the BFT++ implementation.

For experimentation purposes, we implemented a synthetic vulnerability in the PLC. Within the code pathway that receives user-provided (untrusted) input data and commands, we intentionally fail to check the lengths of input strings written to fixed length memory buffers, creating an overflow vulnerability. This is representative of a common class of weaknesses that often lead to remotely exploitable vulnerabilities. When a "malicious" payload is sent to this service, it triggers an attack in the logic PLC with the goal of creating a physically damaging effect on the process apparatus, i.e., causing the VGR to drop the payload the gripper actuator is holding. With this synthetic vulnerability, we can precisely craft a diversification for the desired "fail fast" behavior. To do so, we used a well-established "canary value" technique. The canary is a special guard value inserted between the vulnerable buffer and other sensitive variables in memory and set to a randomized unique value in each of the logic PLCs. Before using the protected variables located after the canary, its value is checked, and if incorrect, the check code simulates a system crash on the PLC.

The transformation just described is sufficient to address the synthetic vulnerability in this experiment. In practice, a more thorough diversification scheme is necessary. The diversity scheme must have sufficient breadth to cover a wide range of potential code weaknesses and sufficient rigor to prevent an attacker from bypassing the scheme. Most importantly, the application of diversification against each of the logic PLCs must make it impossible for the adversary to craft an attack that succeeds against all three simultaneously.

Finally, any diversification transformation must maintain the safety and functionality constraints of the control system. At a minimum, the post-diversification system should be tested to ensure no bugs or adverse behaviors were introduced. This is trivial to manually verify in the experimental apparatus but may be laborious for a more complex system. Ideally, this step should be automated using well-defined test cases that thoroughly exercise the control system's code and behaviors.

4.4.3 Reset and State Recovery Mechanism. The attack detection and failover schemes implemented by the Orchestrator protect the control system from a single instance of an attack. However, they leave the system in a degraded state with two of the three logic PLCs offline or untrusted. A recovery process is necessary to restore the full level of resilience. As with the other elements of BFT++, there are multiple ways to implement this recovery step. The appropriate strategy depends on the attributes of the control logic program, the type of physical processes it controls, and how thorough the reset must be. That said, reset and recovery has two parallel objectives:

(1) Clearing the remnants of a bad input so the affected PLCs can be restored to a trustworthy state to continue operating the process.
(2) Restoring the affected PLCs' ability to correctly control the process before the Orchestrator begins forwarding PLC α's outputs to the hardware again.

Logic Controller Reset. Designing a reset mechanism becomes a complex risk management process. Factors that must be considered include:

- The adversary threat model the system faces
- The control system's exposure and attack surface
- Features and capabilities of the control system devices involved
- Performance of the diversity scheme applied to the logic PLCs
- How long may the physical process operate without a full fault-tolerant set of logic PLCs
- Whether there are "hot spare" logic PLCs available for insertion into the system

Implementing other, non-BFT++, security mitigations can help to narrow the range of threats the BFT++ mechanism is left to address. For example, heavily constraining network access to the logic PLCs can reduce the level of residual risk facing the control system. It should be noted this control may alter the outcome of other risk decisions when designing a BFT++ implementation.

The properties of the diversity mechanism applied to the logic PLCs play the largest role in these decisions. The artificial diversity method is responsible for converting potentially exploitable vulnerabilities into crash bugs. The vulnerability still exists, but a single instance of an exploit should only compromise a single variant; crashing the diversified replicas. Unless the adversary is particularly lucky, most attempted attacks will result in both PLC α and β crashing. In those cases, the attack never succeeded and never could have executed a payload on the logic PLCs to create a malicious effect on the control system. As a result, simply restarting the crashed logic PLCs is thorough enough of a reset to handle this category. When the amount of entropy introduced by the diversity mechanism is high, the chances of a successful compromise of at least one logic PLC is very low. If that chance is low enough to satisfy the implementer's risk tolerance, then one might consider such a simple reset mechanism to be sufficient.

When there is less entropy, hence a smaller search space for the adversary, the chances rise of one PLC being successfully compromised while the others crash. A more thorough reset is necessary for a compromised PLC. It is not possible to rule out that the adversary had time to execute other actions to create a persistent foothold. In the worst case, a thorough reset may require wiping and reinstalling the device's firmware and the control logic program, a lengthy and disruptive process.

Additional confounding factors arise in the low entropy case. If the BFT++ implementation is using a lightweight reset or reboot after PLC α and β both crash and the adversary is aware of this, the adversary might be able to exploit the implementer's assumptions to their benefit. Consider the case where the adversary defeats the diversity on one of PLC α or β. With awareness of a BFT++ implementation using the lightweight reset approach, the adversary could install a backdoor and then simulate a device crash. Such a BFT++ implementation that overestimates the effectiveness of diversification, could then be tricked into thinking the attack failed and only do a lightweight reset. Later, the backdoor or other latent malicious code might activate granting a patient adversary the ability to disrupt the control system.

If additional logic PLCs are available, there are additional options. Increasing the number of diversified PLCs in play would further reduce the probability a single exploit compromises all non-delayed PLCs. Furthermore, it would enable recovery so long as at least two delayed PLCs and one non-delayed remain. If a full level of redundancy is maintained, a compromised device that requires a thorough reset could be taken offline to do so without a lengthy reset process obstructing recovery of the control system. If availability of spares enables this approach, it may even be desirable to isolate and preserve the compromised device for later forensic examination.

Control Loop State Recovery. There are several process engineering considerations involved in selecting a recovery mechanism. These are based on how stateful the process logic is and what consistency constraints must be adhered to while manipulating that state data. These factors depend on the type of physical process being controlled, the capabilities of the sensors and actuators implemented in the control system hardware, and the semantics of the control logic program governing the system. The overall objective of recovery is to manipulate PLCs α and β such that either can immediately, and transparently, retake control of the control system process as if no attack ever occurred. Except in limited cases, this will likely involve identifying and copying some state information from PLC γ to α and β.

In the simplest case, the process control logic may be largely stateless. That is, all the input data necessary to compute the next set of outputs to the physical system is collected from sensor measurements at the beginning of that computation cycle with no dependency on "remembering" what occurred in previous cycles, such as a simple thermostat. This a basic system requires no long-term memory once it is executing its core control loop. With no long-term state to manage, adding BFT++ to a stateless system is relatively simple and no process-specific recovery mechanism is necessary. The other components of the BFT++ solution can be minimally complex and highly abstracted from the details of the process.

Many control systems are more complicated and do require some consideration of state. More complex algorithms might account for past behavior of the system on previous cycles to fine tune control outputs. An example of this is when a control system attempts to gradually approach a set point without overshooting. While such techniques might lead to more efficient operation of the process, it may not be strictly necessary to recover non-critical state information. That is, at the cost of efficiency or smooth operation the process will still operate safely and successfully if a freshly reset PLC is placed in charge of the system with no memory of previous cycles.

Another step up in complexity is systems that have multiple operating "modes" for different conditions like "startup", "initialization", "running", and "shutdown". Control systems like this often have a prescribed sequence of transitions through these states that ensures the process is in a known and safe state. A basic instantiation of BFT++ could utilize this built-in "fail-safe" system behavior. In the VGR case, it is possible to recover the arm to a known "home" position and then return to the desired target position. However, reverting to fail-safes can be time consuming. While a fail-safe sequence executes, it prevents the process from performing its intended duties, a potential denial of service condition, trading system performance for ease of implementation of the BFT++ solution.

Ideally, BFT++ enables a system to continue running without interruption, despite being subject to attacks. That is, the goal is to keep a running process in the "run" mode throughout failover, reset, and recovery steps. The recovery process must be able to restore PLCs α and β directly into the correct mode and bypass any startup sequence that might be present in the logic program. Throughout an attack and recovery sequence, the protected PLC γ will have stayed in the "run" mode and kept the physical process in an equally safe operational state. At a minimum, a BFT++ implementer must: 1) analyze the control logic to understand the system's operating modes, 2) document and identify the program variables responsible for tracking the current mode, 3) extract these values from PLC γ, and 4) inject these values into PLCs α and β following their reset.

More complex control systems will inevitably have more complex control logic with a greater number of internal state machines and state variables. As before, recovering such a system involves completely and consistently copying all this state data from PLC γ to PLCs α and β, and will necessarily be a more invasive process than those described previously. The first and most important step for the BFT++ implementer is to thoroughly analyze the control system logic to identify and understand the states of the system, the flow of execution through the program, the transitions between states and modes, and all persistent state variables. In the VGR example, examination of the control logic program reveals it consists of several interacting state machines that process and execute a sequence of operations that move the arm while picking up and placing objects with the vacuum gripper manipulator.

Early prototypes of the VGR's control program utilized a hard coded sequence of actions. Each step of this sequence, encoded as a state machine, contained many temporary variables and completion flags for marking when the various state transition criteria are complete, e.g., when a motor has finished moving the arm to a specific location; however, these were not essential to the operation of the control loop. That is, within the confines of each step of the sequence, the system acted like the stateless case and the only persistent state data that needed to be copied for the recovery process reduced to the single variable that tracked which step of the sequence the arm was presently executing. More complex systems, with a greater amount of functionality and input modes, have complex state machines and many variables governing those state machines; however, the analysis process is the same. The current version of the VGR program described above has significantly more state than a single sequence counter including, but not limited to, ephemeral state such as the user input string, the current execution progress in that string, the homing status of each arm axis, and the status of each actuator output (motor direction controls, compressor activation, vacuum solenoid status, etc.), as well as more durable state like the calibration lookup table of arm destination positions in the arm axis coordinate system.

Once the critical state variables are identified, the implementer's next step is to devise a strategy for transferring that data to the recovered PLCs. The team experimented with two approaches in the vacuum gripper robot testbed:

- State Transfer — Each logic PLC is allowed to execute independently. Upon reset and recovery, the state from PLC γ is extracted and copied to PLCs α and β. The trio are then allowed to freely execute independently again until the next attack detection.
- State Synchronization — On completion of each execution cycle, the logic PLCs copy their critical state variables to the Orchestrator, along with the rest of their output signals directed at the process hardware. The Orchestrator then selects one set of these variables and redistributes them to each logic PLC for use at the beginning of their next execution cycle.

State transfer is generally the preferred approach. It has the advantage of not requiring trust in any data from PLC α and β after the moment attack detection is triggered, thereby minimizing the effects of any changes an adversary might have been able to perform before detection triggered. PLC γ takes over the process control, using its own state information maintained from executing in parallel with the other logic PLCs, albeit from behind the delay queue, without any dependency on what PLC α and β did in the past. The recovery mechanism establishes a connection to all three logic PLCs using the Siemens S7 management protocol, copies the state variables from their location in PLC γ's memory using its standard functionality, and then writes the variables into the corresponding locations in PLC α and β. A potential downside to this approach is that because of the delay queue, and precisely what information is delayed, PLC γ may have a slightly out of sync internal representation of the process compared to the non-delayed PLCs leading to rough or jarring failovers from one PLC to another.

State synchronization may be useful in cases where all three logic PLCs must be kept in stricter alignment. Using the same PROFINET data transfer mechanism leveraged by the Orchestrator to proxy the logic PLCs' inputs and outputs, each PLC includes its state data (the state machine sequence counter variable) along with the control loop's outputs that are sent to the Orchestrator at the end of every execution cycle. The Orchestrator then chooses one set of the state data to redistribute to each logic PLC for use at the beginning of their next execution cycle. This keeps all three PLCs better synchronized over time and prevents any potential drift that might occur if they ran independently. However, if the current primary PLC is successfully compromised by an attack, there is a small window for the attack's effects on that PLC to be replicated to the otherwise protected PLC γ via the state synchronization. The risk created by this will vary depending on the reaction time of the attack detection and should be appropriately weighed against the potential benefits of stronger synchronization before considering this approach.

Whichever strategy is chosen for state transfer, another challenge to consider is maintaining the consistency and integrity of the data. Copying data from one PLC to another is not an instantaneous atomic action. It takes time, during which the PLC may continue executing its program. Depending on the PLC and network protocols involved, while the transfer is in progress but not yet complete, the data may be in an inconsistent state on the destination PLC. This could potentially occur even in a system as simple as early versions of the vacuum gripper robot instantiation where the state to synchronize consisted of only a single multi-byte sequence counter integer. The semantics of the Siemens S7 protocol we use for state data transfer do not guarantee the consistency of the bytes being written until the transfer has fully completed, so even a 2- or 4-byte integer may be impacted. The more complex, user input driven demonstration application contains a much greater amount of state, increasing the likelihood of consistency problems. Even with a low likelihood, in a long-running control system with a larger amount of state data, inconsistency is likely to occur eventually and corrupt the safe and correct system operation. If the PLC and network protocols used allow for guaranteed consistent data transfers, then proper use of those features solve this problem. If not, it falls on the implementer.

One possible solution for this situation, which we implemented, is to hold PLCs α and β in a stopped state until the state recovery data transfer is complete, and only then activate them to begin executing the control logic loop. At the beginning of the reset and recovery process, the Orchestrator activates a flag in both PLC α and β to stop them executing, like the simulated "crash" state. The process then continues through resetting the devices to clear the attack, writes the state recovery data into both PLCs, and finally clears the stop flag to allow them to begin executing again. This arrangement acts as a lock to ensure no code is attempting to access the state variables during the duration of their transfer.

Another consistency consideration is how the state variables are affected by the "fail fast" brittleness mechanism used for initial attack detection. A common technique used to achieve this is artificial diversification of the device software and data structure layout. This might include altering — even randomizing — the location of data variables in memory so that an attacker will not know their precise location across differently diversified devices. In a PLC program, this might take the form of altering the memory address, data block number, or other location specifier of a given state variable. Whichever diversification is performed, the state recovery mechanism must incorporate a map indicating the correct locations of each state variable in the source and destination PLCs.

The state transfer and recovery processes are unavoidably invasive and, if not carefully implemented, may jeopardize the safe and error-free operation of the control system. Therefore, it is imperative that the implementer thoroughly test and validate the failover and recovery process. This should include testing each operating modes and state of the control system, especially rarely occurring states such as fail-safes and error handlers.

4.4.4 Gateway Proxy / Delay Queue. The delay queue is the primary mechanism protecting PLC γ from attacks. Attack detection serves to command the queue to drop its potentially malicious contents, while reset and recovery only matter if the queue succeeds at its job. As with the other instantiation components, the queue's architecture is influenced by the design of the control system. The queue mediates the communications and network attack surface of the logic PLCs, so this surface must be fully enumerated. Necessary information includes the type of network interfaces and protocols in use plus the type of messages, packets, and data contained therein. As a baseline, firewalls and other common means should be applied to restrict the scope of connections and protocols. In addition to being good security practice, this will also simplify construction of the delay queue.

Packet and Frame-level Approaches. In early prototypes of our BFT++ implementation we experimented with delay queue implementation that operate at the lower layers of the network stack. Each operated as an in-line bridge between the Ethernet network switch and PLC γ. In the first case, user datagram protocol (UDP) datagrams destined for PLC γ are extracted from the bridge, placed into a delay buffer, then rebroadcast when each packet had been delayed for the desired time. The second case performed much the same way but operated on Ethernet frames instead of UDP packets. If this simple approach were to work, it would have the benefit

of being largely agnostic to the higher-layer application protocol contents traversing the queue. For a simple case, e.g., a custom UDP protocol we added to the vacuum gripper robot's control logic program, it performs well. However, it quickly fails for more complicated protocols. For instance, anything that implements automatic retransmission of dropped packets, like transmission control protocol (TCP), cannot be queued and flushed this way.

Presenting a Unified Network Attack Surface. For the BFT++ protection to work, the set of logic PLCs must receive all network traffic simultaneously and appear to the network outside of the BFT++ implementation as if they were a single unified device. Inbound "untrusted" network messages and connections should be directed to a single address and then replicated to each of the redundant logic PLCs simultaneously. If the logic PLCs can be individually addressed, then an adversary may arrange their attacks to bypass the protections. E.g., by directly targeting and compromising PLC α without crashing PLC β, an adversary could achieve whatever physical effect they had planned.

A packet replicator can handle simple cases to achieve the unified surface property, but, as complexity increases, it may quickly encounter problems. In the simplest cases where devices in a particular network bus are not individually addressed and all traffic is broadcast, then a simple replicator may be sufficient. This may work for inbound messages, but more complex logic is necessary to ensure that the set of logic PLCs are sending consistent, non-conflicting responses back onto the network. A similar arrangement to the I/O multiplexing responsibility of the Orchestrator could be used. For the larger class of protocols that involve individual device addressing, some form of network address translation (NAT) is necessary. The unified surface mechanism must still replicate packets destined for each logic PLC but must appropriately rewrite any address fields so that the PLCs know how to appropriately handle the messages.

For richer, more complex communications, NAT is insufficient and protocol-aware NAT or protocol-aware proxies may be necessary. Because they share similar requirements, it can be advantageous to combine the delay queue and network surface unification functions into a single mechanism with a design analogous to the kinds of protocol-aware load balancers and reverse proxies often found in scalable enterprise services.

Protocol-aware Reverse Proxy Approach. We ultimately implemented a unified protocol-aware reverse proxy and delay queue system. This construct sits in front of the BFT++ system and acts as the sole external network interface for the control system. Outside users (and adversaries) may only interact with the vacuum gripper robot via the single IP address and protocol interface exposed by the proxy and are blocked from directly interacting with the logic PLCs or other internal components of the BFT++ enabled control system. To demonstrate how a BFT++-enabled system would integrate with wider world (SCADA systems, operator consoles, etc.) we chose OPC UA as the proxy's protocol. Different makes and models of PLC could be swapped into the logic control, or even mixed and matched for a further level of diversity, demonstrating the extensibility of our implementation.

The proxy implements an OPC UA server containing a data model for the inputs and outputs of the vacuum gripper robot's external interface. This includes data like the current positions of the arm axes, if a command is executing or finished, whether the motors are running and in which direction, and similar data that would be useful to display on an HMI. It also has fields for inputting a movement command string and starting its execution. These are translated by the proxy into write commands to locations in the logic PLCs' data memory. Instead of a delay queue of packets or frames, the proxy's queue operates on these abstracted read and write messages where they are held for a time before being released to send to PLC γ. The proxy's backend interface to the logic PLCs processes these abstract messages and converts them to equivalent read and write commands in the S7 management protocol which are sent simultaneously to the non-delayed logic PLCs. The proxy / queue construct monitors a digital output signal from the Orchestrator indicating when a queue flush should take place, eliminating any still unsent writes to PLC γ.

5 EVALUATION RESULTS

The protection afforded by BFT++ comes down to timing. That is, can the system detect an attack and react quickly enough to flush the delay queue so PLC γ stays protected. Certain variables are dictated by the implementation and physics of the process and hardware; however, several are under the operator's control. The challenge is to tune the system as close as possible to the physical limits of the process (with some safety margin), to minimize the reaction time of the attack detection and failover mechanism, and maximize the resilience imparted by BFT++.

The main components of the timing calculation are as follows. Table 2 provides a summary of measures.

- Post-attack PLC crash delay. How long it takes for logic PLC (α or β) to crash following the attack. This time is determined by how effective the fail-fast diversification is for the logic PLCs. Our experimental implementation using a canary value converts this style of overflow weakness into a crash nearly instantaneously (Table 2e), but other implementations may have different characteristics that must be understood for optimal tuning.
- Crash detection delay. The time it takes for the Orchestrator to recognize one of PLCs α and β has crashed. In our experiment, the running length of the watchdog timer is the dominant component here. As the counter value is passed to the Orchestrator, the network latency adds to the overall detection time; however, this will only be a significant factor on unusually slow networks. Execution time of the logic is another factor (Table 2b and d); however, in almost all cases, it will be negligible compared to other components.
- Failover delay. How long the Orchestrator takes to switch which control logic outputs are forwarded to the physical apparatus remote I/O. In our experiment, the Orchestrator decides which set of control signals to forward to the hardware I/O every execution cycle. The protected PLC γ is continually running in parallel with the primary pair, and there is no delay in its outputs taking over control of the physical system. Following detection of an attack, the Orchestrator will immediately switch the "in control" logic PLC to γ (Table 2g).
- Delay queue flush interval. How long it takes to flush the message queue. Like the failover decision, sending the flush command happens instantaneously with crash detection. The interval is hence reduced to the time for the delay queue mechanism to receive and fully execute the flush command. This should be optimized, but is otherwise independent of the details of the control system.
- Physical inertia time budget. How long the physical system can safely operate without control; essentially how long inertia can absorb a fault. This sets the maximum allowable reaction time, i.e., the sum of previous delays, for the BFT++ protection mechanism. This time will vary for each physical system (Table 2h).

Additional delays do not affect initial responsiveness to an attack, but do impact how long the system remains in a degraded state where it cannot mitigate an additional attack:

- Reset and recovery delay. How long it takes to reset logic PLCs α and β to an uncompromised state and clear any residual data. Depending on the thoroughness of the reset, this may be substantial. Operators and defenders of the physical system must balance this carefully.
- Logic PLC startup delay. How long, after completion of a reset, until the control loop on the logic PLCs is fully operational and liveness values are updating again. This is dependent on the reset and recovery implementation chosen and the architecture of the control logic.
- Liveness detection and switchover delay. The time it takes the Orchestrator to recognize PLCs α and β are functional again and switch control of the physical process from γ back to α, the inverse of the failover delay plus the latency of liveness value transmission.

The remainder of this section describes the experiments and measures performed to optimize these variables to achieve the best attack protection success rate. We instrumented the Orchestrator, Proxy / Delay Queue, and logic PLCs to characterize the timing and behavior of the implementation.

Table 2. Measured timing components (milliseconds).

	Min	Max	Avg	StdDev	99 %	99.9 %	99.99 %
a) PLC α write latency (S7-1500)	1.048	13.85	1.386	0.467	2.205	3.239	4.293
b) PLC α cycle time	0.126	2.932	0.403	-	-	-	-
c) PLC β write latency (S7-1200)	5.36	755.566	9.583	3.575	21.255	28.697	42.463
d) PLC β cycle time	0.269	2.508	0.874	-	-	-	-
e) Crash delay	Within 1 execution cycle of canary value overwrite on logic PLC						
f) Crash detection	Within 1 execution cycle of watchdog timer expiration on logic PLC						
g) Failover delay	Within 1 execution cycle after crash detection on orchestrator PLC						
h) Gripper inertia	200ms minimum. Unreliable beyond 250ms from physical imperfections.						

5.1 Process Inertia and Control Loss Tolerance

Determining the amount of physical inertia in the process is a key element of implementing and tuning a BFT++ instantiation. The simplest approach to this relies on system documentation, specifications, and expert knowledge from engineers and operators familiar with the system. Note that operator experience alone may not be sufficient to characterize all mechanics in sufficient detail. Another straightforward option is to empirically test the operational system or a representative testbed. When a testbed is not available, models or "digital twins" of the system can be used. However, the models must have sufficient fidelity to capture the physics-based responses of the system over the whole range of states and inputs. Finally, empirical testing on a production system is possible, but considered a last resort.

For experimentation, we utilized empirical tests on the factory simulation; a low-risk physical model of a manufacturing line. This allowed us to perform tests too dangerous with real industrial robotics and factory equipment, e.g., losing control over arm movement or dropping its payload. While harmless in the model, either failure mode is a significant hazard when scaled up. The motor and gear-driven movements of the robot arm are also worth consideration. However, each motor operates via a simple on-off switch leaving little inertia to exploit.

The vacuum gripper is more interesting, and the focus of our investigation. Attacks seeking to manipulate the gripper actuator may attempt to prevent it from picking up a load or cause it to drop a load at an undesirable time. The latter case presents an opportunity to leverage physical inertia. The mechanics of the pneumatics and the time needed for gravity to accelerate a load away from the gripper provide a significant amount of time before a load is irretrievably dropped. Manual measurement with a small test PLC program showed at least 200ms (Table 2h) to be the maximum time the gripper could be deactivated while reliably being able to catch the payload again. Mechanical imperfections reduce the chance of a successful catch for longer deactivations.

5.2 Attack Response Reaction Time

The ability of the BFT++ mechanism to reliably protect a control system depends on how quickly it can detect an attempted attack and flush the delay queue, thus discarding the malicious input data before it reaches PLC γ. This primarily involves two variables, the delay queue size and crash detection watchdog timer threshold. To prevent an attack message from reaching PLC γ, the message must still be held in the queue when it is flushed. The queue delay must be at least as large as the total attack reaction time. The largest contributors to reaction time are the crash detection delay, dominated by the crash detection watchdog timer length, followed by the

delay queue flush interval and other implementation dependent latencies. An optimal configuration will seek the minimum possible watchdog timer and maximum delay queue size.

5.2.1 Configuring the Crash Detection Timer. The Orchestrator uses a timeout-based mechanism to detect when one or more logic PLCs have failed. All the logic PLCs increment a liveness counter on each execution cycle, which is then transmitted to the Orchestrator. Failure to update this value initiates a watchdog timer. The upper and lower bounds for this timer duration must be characterized for a given instantiation of BFT++. Too low will result in false positive crash detections due to network jitter, benign packet loss, etc. when a logic PLC is operating nominally. Too high eliminates false positives but reduces the responsiveness to attacks, exceeding the process inertia determined in Section 5.1 and potentially allowing attacks to negatively effect the physical system.

False positive crash detections can occur when the Orchestrator's timeout is too sensitive. There are several ways this can lead to a false positive:

- If a message containing an updated value is corrupted or lost
- If network latency significantly delays an update message
- If a logic PLC's execution cycles take significantly longer to execute than the Orchestrator's

If any factor leads to a liveness update failing to register before the Orchestrator's timeout expires, then a logic PLC may be falsely marked as crashed. However the minimum timeout is calculated, the chosen value should account for all possible legitimate behaviors of the system. These include: the maximum possible execution cycle time any of the logic PLCs might encounter, the similar longest possible legitimate execution cycle of the Orchestrator, and potential worst case network conditions that the control system must endure. Our measurements show setting the watchdog timeout value to 6ms or more yields no false positives[2], but they rapidly increase in frequency for shorter timeouts, reaching a limit that strongly correlates with network latency. Finally, in addition to the ceiling provided by the inertia calculation, the maximum crash detection timeout value must be less than the time inputs are held in the delay queue before being delivered to PLC γ. Otherwise, queue flushes would occur too late to prevent all three logic PLCs from being affected.

5.2.2 Delay Queue Duration. Determining the minimum delay queue duration is relatively straightforward and depends mostly on the implementation of the BFT++ mechanism, while being largely independent of the control system implementation. At its absolute minimum, the duration for which the delay queue holds messages before delivery must exceed the time it takes for the BFT++ mechanism to detect an attack attempt and successfully flush the message queue. Characterizing the maximum delay queue is more complicated and involves a wider range of factors.

One must first determine how sensitive the control system is to input message latency. This may depend on both the network protocols in use, the semantics of the input data traversing those protocols, and the nature of control and physical system. In the case of the movement command strings entered via the OPC UA proxy / delay queue frontend, there is no strict requirement for how quickly the system executes those commands. The TCP-based S7 management protocol is similarly tolerant of delays as high as 5 seconds even when device programming and similar tasks are routed through a delay queue. However, delaying real-time protocols such as PROFINET poses a greater challenge. We observed that delays of 500ms or more causes PROFINET connections between PLCs to become inoperable. It is also intolerant of packet loss caused by a queue flush. In the default configuration of PROFINET for the S7-1200 and S7-1500 PLCs in our testbed, flushing 3 or more consecutive packets causes a link disconnection error, causing a multi-second interruption as the connection is reestablished.

The delay queue length must also account for the time needed to send the input and for it to be received by the target PLCs, when the malicious effect begins to take place, and the setting of the attack detection watchdog. The latter are discussed in the previous Section, but one must also consider how long it takes for the proxy /

[2]Measured over multiple trial runs; each of which over 60-minutes long

delay queue to deliver the potentially malicious input messages to PLCs α and β. The proxy implementation uses the S7 protocol to write the input data into the logic PLCs' memory. This takes a variable, and potentially long time as shown in Table 2a and 2c. Clearly, we cannot set the delay queue length large enough to accommodate the slowest possible write to PLC β ($755ms$), as it exceeds the inertia window, so we must accept a smaller, more reasonable delay that may incur some false negatives. This is an unavoidable trade off that must be balanced against the risk tolerance of the system operator.

5.2.3 Queue Flush and Other Latencies. The remaining timing components contributing to success or failure (post-attack crash delay, failover delay, and delay queue flush time) proved through measurement to be make a negligible contribution to the overall timing budget.

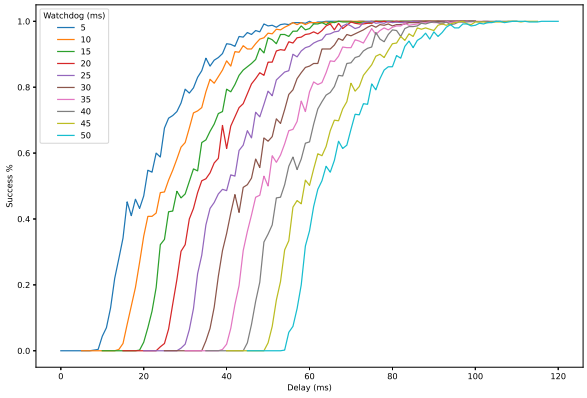
5.2.4 Reaction Time vs Queue Size. It remains to explore the effect of the delay queue and watchdog timer. It is desirable to set the watchdog timer as low as possible without incurring false positives. We tested settings as small as 5ms, but to avoid false positives we typically use a 10ms minimum. Holding this variable constant, we tested a range of queue delays between $watchdog - 5ms$ to $watchdog + 70ms$ in $1ms$ increments, with 500 trials at each setting. While queue delays smaller than the attack detection watchdog timer would never be used in practice, it proved useful to illustrate some behaviors of the system at its limits. The third tested variable was which logic PLC the attack was directed at: compromising PLC α and crashing β, crashing PLC α and compromising PLC β, and crashing both (the most likely case in operation unless the adversary has foreknowledge of the diversification). Figure 5 shows the results of these tests.

The first observation is that different watchdog timers result in roughly the same curve shifted left or right. As shown in Figure 5a, when PLC β, the slower S7-1200 device, is one that crashes and triggers the flush and reset process, it takes a queue delay of approximately 5ms higher than the crash detection watchdog timer before successful trials begin to occur and the graph begins to rise. The success rate climbs slowly before approaching 100% success at over 40ms greater than the watchdog timer. Each graph shows that even when approaching the maximum tested queue delay values, the success rate for each watchdog setting is not reliably 100%. We traced these sporadic failures to the often erratic write-latencies observed in Section 5.2.2. The jaggedness of the graphs demonstrate the significant variance in the network write times for PLC β.

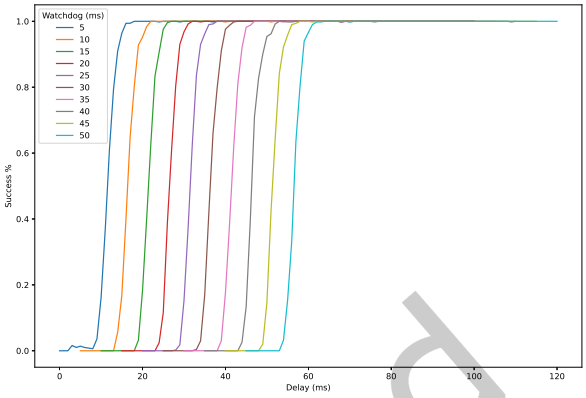
Figures 5b and 5c show the case where PLC α is crashed (with or without compromising β). The faster PLC α (S7-1500), exhibits much quicker and more reliable network write times. This results in the attack data reaching the PLC more quickly than in the prior case, and with far less variance. This results in success rates that approach 100% more quickly, in less than 15ms greater than the watchdog setting. Once at this peak, the success rate only occasionally dips below near 100% success and for only a handful of trials per setting combination.

These findings argue in favor of choosing devices with similar performance characteristics for the three logic PLCs. In this case, the additional hardware diversity gained from mixing S7-1500 and S7-1200 devices may not be worth the additional variance and absolute performance impact in the attack protection success rate.

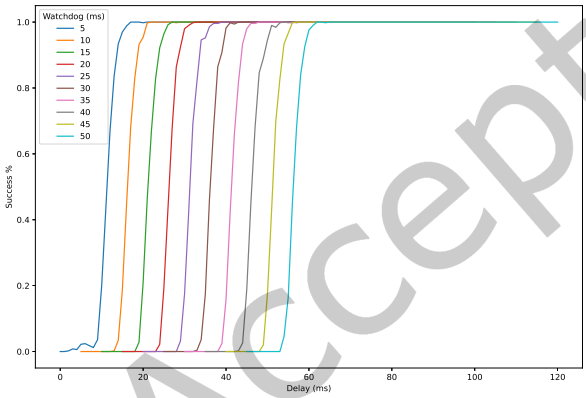
Another interesting feature of the data is what occurs in the $watchdog = 5ms$ case when the PLC α, the S7-1500, is primarily triggering crash detection. As can be seen in Figures 5b and 5c, the 5ms plot shows a number of successful trials for queue delays we would reason are too short for any success to occur, and for larger watchdogs, similarly small queue delays indeed have 0 successful trials. Inspecting the raw timing measurement data showed that cases exhibited "negative" crash detection reaction times. That is, subtracting the watchdog timer setting from the measured crash time resulted in a timestamp prior to when the logic PLC actually entered the crash state. We traced this to an artifact in how the crash detection watchdog timer logic works. As that logic works by continually transferring a counter value from each logic PLC to the Orchestrator, differences in the cycle execution times of each PLC in the arrangement can lead to cases where the Orchestrator executes two cycles faster than one of the logic PLCs can. This appears to the orchestrator as a false positive activation of the watchdog timer. The 5ms watchdog case is close enough to the sum of the cycle execution times and network



(a) Compromised PLC α, Crashed PLC β



(b) Compromised PLC β, Crashed PLC α



(c) Crashed PLC α and β

Fig. 5. Attack Response Reaction Time (watchdog vs. delay).

latency that we observed the watchdog timer beginning its count down with great frequency. This effectively reduces the watchdog timer setting by several milliseconds, allowing certain "impossible" cases of trial success. This same effect is not observed in Figure 5a, where the S7-1200 PLC β triggers crash detection. We believe this is because the effect just described is buried by the success rate noise created by its much higher data write times.

This finding illustrates the importance of characterizing all the elements of the BFT++ system, including the physical apparatus, the devices implementing the control logic and BFT++ mechanism, and the network upon which they reside. Each introduces variables that must be accounted for to predict how the final BFT++-enabled control system will behave in the face of attacks.

6 RELATED AND FUTURE WORK

6.1 Stopping Repeat Attacks

So far, our plan for cyber resilience has allowed older control systems to identify cyber attacks during their normal operation, automatically triggering a quick and efficient recovery process. However, there is still a concern that attackers may exploit this system behavior to launch an availability attack. One major limitation of our

implementation is the inability to handle rapid follow-up attacks. Our delay queue implementation does not have a “memory” of previously seen bad packets. Therefore, if a bad message is quickly replayed, the attack will reach PLC γ, and crash (or compromise) it. Preventing this situation is conceptually easy, but requires further work. First, the offending messages in the delay queue must be identified. Second, the system requires a bus-level blacklist filter that remembers and filters malicious inputs identified from previous crash behavior. By mediating bus traffic, reusing a previous exploit can be prevented. Solving these challenges is a subject of our ongoing efforts.

6.2 Single Controller Variation

YOLO [2, 3] is another variant of BFT++ used as a secure resiliency mechanism tailored for CPSs. Theoretical foundation for YOLO has been analyzed in the following articles [9, 11, 20–22], while an analysis for the overall BFT++ family of CPS resilience methods can be found in a recent article [18]. YOLO combines the techniques of reset and diversification which allow it to be simpler in its implementation, as well as less intensive while simultaneously providing significant defence against foreign cyber attacks. In a traditional system, frequent resets will degrade the usability of the system. In CPS, however, the physical subsystem can continue to move and operate even during resets because of its inertia which enables YOLO to be a simpler to implement and low-resource alternative to the original BFT++. However, the YOLO method results in a probabilistic approach to CPS security, and does not give the same security guarantee as the approach shown in this paper.

6.3 Subprocess BFT++

Another variation of the BFT++ model is currently under development. While all previous variants of BFT++ operate on the whole control program, this approach, called called SubBFT++, operates on the subprocesses. This would leverage the advantages of both the original BFT++ and the YOLO variant to achieve a protection level equaling the original BFT++ but with the efficiency of YOLO. SubBFT++ would analyze the system code, function by function, and identify parts that require higher levels of protection of vanilla BFT++ (diversified replication), and parts that can sufficiently be protected by YOLO (diversification only).

Since SubBFT++ works on the subprocess level, the operator can configure and trade off the level of protection against operational/execution cost. Cost and protection level tuning capability will be incorporated into SubBFT++ configuration toolset, allowing user to specify the slack time currently available to the system as well as the minimum slack time the user would want the system to operate on. Based on the user’s inputs, the algorithm of SubBFT++ would examine all functions in the system and identify the appropriate protection technique for each of the functions without violating the cost and performance bounds set by the user.

6.4 Virtual Inertia

Also under work is the use of other phenomena in place of physical inertia. Generalizing inertia as "the resistance to a large change in state in a small amount of time", BFT++ can be extended to systems without explicit physical inertia in two ways. First, systems that display or otherwise proxy physical systems (e.g., radar displays, LIDAR displays) exhibit "inherited inertia". For example, a radar display does not have inertia itself, but the objects it tracks do. Therefore, elements of the radar system inherit the inertia of the objects they sense. Secondly, other systems exhibit phenomena similar to physical inertia. For example, streaming audio and video are able to absorb short faults (dropped packets) without significant impact to the end user. Further exploration is necessary to determine if BFT++ can indeed be extended to these systems as well.

7 CONCLUSIONS

Overall, this effort successfully demonstrates application of BFT++ to a control system with real-world controllers in a industry-relevant setting. General purpose automation controllers of this sort are found in many operational technology and control systems environments including manufacturing, building automation, ship mechanical systems, and other industrial systems. We have shown how to retrofit BFT++ onto an existing control system without significant modification. Furthermore, the resulting system is highly effective at protecting against simulated attack scenarios for which the application of diversification can create a divergence in attack behavior, e.g., memory safety vulnerabilities.

We have also shown how to implement BFT++ within the confines of a Siemens PLC environment. Much of this was in the practical design of the system and application of device features to implementing the scheme described in Section 5. We initially built the non-BFT++ implementation without consideration for how to apply BFT++, mimicking how a retrofit would actually occur on a real-world system.

We also conclude that certain choices during the original process design and implementation can make applying BFT++ easier. Many of these choices relate to synchronizing and recovering state across failover and recovery operations. When possible, we would suggest co-designing the BFT++ implementation along with the process logic, and if possible, even along with the control system hardware, particularly with regard to adding more sensors that allow the system to monitor its own state directly via sensor measurements rather than indirectly via extrapolation.

One major limitation of our implementation is the inability to handle rapid follow-up attacks. Our delay queue implementation does not have a "memory" of previously seen bad packets. Therefore, if a bad message is quickly replayed, the attack will reach PLC γ, and crash (or compromise) it. Preventing this situation is conceptually easy, but requires additional effort. First, the offending messages in the delay queue must be identified. Second, the system would require a bus-level blacklist filter. Solving these challenges is a subject of our ongoing efforts.

REFERENCES

[1] Riham Altawy and Amr M. Youssef. 2016. Security, Privacy, and Safety Aspects of Civilian Drones: A Survey. *ACM Trans. Cyber-Phys. Syst.* 1, 2, Article 7 (nov 2016), 25 pages. https://doi.org/10.1145/3001836

[2] Miguel Arroyo, Hidenori Kobayashi, Simha Sethumadhavan, and Junfeng Yang. 2017. FIRED: frequent inertial resets with diversification for emerging commodity cyber-physical systems. *arXiv preprint arXiv:1702.06595* (2017).

[3] Miguel A Arroyo, M Tarek Ibn Ziad, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. 2019. YOLO: frequently resetting cyber-physical systems for security. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, Vol. 11009. International Society for Optics and Photonics, 110090P.

[4] Carmen Camara, Pedro Peris-Lopez, and Juan E. Tapiador. 2015. Security and privacy issues in implantable medical devices: A comprehensive survey. *Journal of Biomedical Informatics* 55 (2015), 272–289. https://doi.org/10.1016/j.jbi.2015.04.007

[5] Ilias Giechaskiel and Kasper Rasmussen. 2019. Taxonomy and challenges of out-of-band signal injection attacks and defenses. *IEEE Communications Surveys & Tutorials* 22, 1 (2019), 645–670.

[6] Ilias Giechaskiel, Youqian Zhang, and Kasper Bonne Rasmussen. 2019. A Framework for Evaluating Security in the Presence of Signal Injection Attacks. *CoRR* abs/1901.03675 (2019). arXiv:1901.03675 http://arxiv.org/abs/1901.03675

[7] Jairo Giraldo, Esha Sarkar, Alvaro A. Cardenas, Michail Maniatakos, and Murat Kantarcioglu. 2017. Security and Privacy in Cyber-Physical Systems: A Survey of Surveys. *IEEE Design & Test* 34, 4 (2017), 7–17. https://doi.org/10.1109/MDAT.2017.2709310

[8] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. 2018. A Survey of Physics-Based Attack Detection in Cyber-Physical Systems. *ACM Comput. Surv.* 51, 4, Article 76 (jul 2018), 36 pages. https://doi.org/10.1145/3203245

[9] Paul Griffioen, Raffaele Romagnoli, Bruce H Krogh, and Bruno Sinopoli. 2019. Secure networked control via software rejuvenation. In *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 3878–3884.

[10] Song Han, Miao Xie, Hsiao-hwa Chen, and Yun Ling. 2014. Intrusion Detection in Cyber-Physical Systems: Techniques and Challenges. *Systems Journal, IEEE* 8 (12 2014), 1049–1059. https://doi.org/10.1109/JSYST.2013.2257594

[11] Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2020. Software fault tolerance for cyber-physical systems via full system restart. *ACM Transactions on Cyber-Physical Systems* 4, 4 (2020), 1–20.

[12] Marek Jawurek, Florian Kerschbaum, and George Danezis. 2012. *Privacy Technologies for Smart Grids - A Survey of Options*. Technical Report MSR-TR-2012-119. https://www.microsoft.com/en-us/research/publication/privacy-technologies-for-smart-grids-a-survey-of-options/

[13] Maryna Krotofil and Dieter Gollmann. 2013. Industrial control systems security: What is happening?. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. 670–675. https://doi.org/10.1109/INDIN.2013.6622964

[14] Francesco Liberati, Emanuele Garone, and Alessandro Di Giorgio. 2021. Review of Cyber-Physical Attacks in Smart Grids: A System-Theoretic Perspective. *Electronics* 10, 10 (May 2021), 1153. https://doi.org/10.3390/electronics10101153

[15] Jing Liu, Yang Xiao, Shuhui Li, Wei Liang, and C. L. Philip Chen. 2012. Cyber Security and Privacy Issues in Smart Grids. *IEEE Communications Surveys & Tutorials* 14, 4 (2012), 981–997. https://doi.org/10.1109/SURV.2011.122111.00145

[16] J Sukarno Mertoguno, Ryan M Craven, Matthew S Mickelson, and Daniel P Koller. 2019. A physics-based strategy for cyber resilience of CPS. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, Vol. 11009. International Society for Optics and Photonics, 110090E.

[17] Robert Mitchell and Ing-Ray Chen. 2014. A Survey of Intrusion Detection Techniques for Cyber-Physical Systems. *ACM Comput. Surv.* 46, 4, Article 55 (mar 2014), 29 pages. https://doi.org/10.1145/2542049

[18] Luyao Niu, Abdullah Al Maruf, Andrew Clark, J Sukarno Mertoguno, and Radha Poovendran. 2022. An Analytical Framework for Control Synthesis of Cyber-Physical Systems with Safety Guarantee. In *IEEE 61st Conference on Decision and Control (CDC)*.

[19] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. 2011. Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*. 2195–2201. https://doi.org/10.1109/CDC.2011.6160641

[20] Raffaele Romagnoli, Paul Griffioen, Bruce H Krogh, and Bruno Sinopoli. 2020. Software rejuvenation under persistent attacks in constrained environments. *IFAC-PapersOnLine* 53, 2 (2020), 4088–4094.

[21] Raffaele Romagnoli, Bruce H Krogh, and Bruno Sinopoli. 2019. Design of software rejuvenation for cps security using invariant sets. In *2019 American Control Conference (ACC)*. IEEE, 3740–3745.

[22] Raffaele Romagnoli, Bruce H Krogh, and Bruno Sinopoli. 2020. Robust Software Rejuvenation for CPS with State Estimation and Disturbances. In *2020 American Control Conference (ACC)*. IEEE, 1241–1246.

[23] Michael Rushanan, Aviel D. Rubin, Denis Foo Kune, and Colleen M. Swanson. 2014. SoK: Security and Privacy in Implantable Medical Devices and Body Area Networks. In *2014 IEEE Symposium on Security and Privacy*. 524–539. https://doi.org/10.1109/SP.2014.40

[24] David I. Urbina, Jairo Giraldo, Alvaro A. Cardenas, Junia Valente, Mustafa Faisal, Nils Ole Tippenhauer, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. *Survey and New Directions for Physics-Based Attack Detection in Control Systems*. https://doi.org/10.6028/NIST.GCR.16-010 QC 20170509.

[25] Guangyu Wu, Jian Sun, and Jie Chen. 2016. A survey on the security of cyber-physical systems. *Control Theory and Technology* 14 (02 2016), 2–10. https://doi.org/10.1007/s11768-016-5123-9

[26] Chen Yan, Hocheol Shin, Connor Bolton, Wenyuan Xu, Yongdae Kim, and Kevin Fu. 2020. SoK: A Minimalist Approach to Formalizing Analog Sensor Security. In *2020 IEEE Symposium on Security and Privacy (SP)*. 233–248. https://doi.org/10.1109/SP40000.2020.00026

[27] Zhiyuan Yu, Zack Kaplan, Qiben Yan, and Ning Zhang. 2021. Security and Privacy in the Emerging Cyber-Physical World: A Survey. *IEEE Communications Surveys & Tutorials* 23, 3 (2021), 1879–1919. https://doi.org/10.1109/COMST.2021.3081450