Trustworthy Embedded Computing for Cyber-Physical Control

Lee Wilmoth Lerner

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor in Philosophy
in
Computer Engineering

January 30, 2015
Blacksburg, Virginia

Keywords: Trustworthy Computing, Secure Computing, Autonomic Computing, Cybersecurity, Embedded Systems, Cyber-Physical Systems, Process Control Systems

Trustworthy Embedded Computing for Cyber-Physical Control

Lee Wilmoth Lerner

ABSTRACT

A cyber-physical controller (CPC) uses computing to control a physical process. Example CPCs can be found in self-driving automobiles, unmanned aerial vehicles, and other autonomous systems. They are also used in large-scale industrial control systems (ICSs) manufacturing and utility infrastructure. CPC operations rely on embedded systems having real-time, high-assurance interactions with physical processes. However, recent attacks like Stuxnet have demonstrated that CPC malware is not restricted to networks and general-purpose computers, rather embedded components are targeted as well. General-purpose computing and network approaches to security are failing to protect embedded controllers, which can have the direct effect of process disturbance or destruction. Moreover, as embedded systems increasingly grow in capability and find application in CPCs, embedded leaf node security is gaining priority.

This work develops a root-of-trust design architecture, which provides process resilience to cyber attacks on, or from, embedded controllers: the Trustworthy Autonomic Interface Guardian Architecture (TAIGA). We define five *trust requirements* for building a fine-grained trusted computing component. TAIGA satisfies all requirements and addresses all classes of CPC attacks using an approach distinguished by adding resilience to the embedded controller, rather than seeking to prevent attacks from ever reaching the controller. TAIGA provides an on-chip, digital, security version of classic mechanical interlocks. This last line of defense monitors all of the communications of a controller using configurable or external hardware that is inaccessible to the controller processor. The interface controller is synthesized from C code, formally analyzed, and permits run-time checked, authenticated updates to certain system parameters but not code. TAIGA overrides any controller actions that are inconsistent with system specifications, including prediction and preemption of latent malwares attempts to disrupt system stability and safety.

Trustworthy Embedded Computing for Cyber-Physical Control

Lee Wilmoth Lerner

DEDICATION

I would like to thank all of my Virginia Tech colleagues who collaborated with me on this work, including: Dr. Mohammed M. Farag, Dr. Yousef Iskander, Dr. William Baumann, and Christopher J. McCarty. I would also like to thank my colleagues and students at the Georgia Tech Research Institute who supported this work, including: Anita H. Pavadore, Ron J. Prado, Zane R. Franklin, Kevin G. Lyn, and Gregory Stein. Additional thanks go to my former colleagues at Luna Innovations Inc. for providing technical insight and the flexibility to pursue my schooling. I am extremely fortunate to have found Dr. Cameron D. Patterson for a Ph.D. advisor. I will be forever grateful for his intellectual guidance, kind encouragement, and friendship. I am also grateful to the remainder of my Ph.D. committee for their guidance. Finally, I am thankful for support from my family, especially: my grandparents, Alwyn and Karen Wilmoth, for the inspiration and heartfelt support; my mother, Brenda W. Lerner, for her good wishes; my brother, Brian J. Cafferty, for the shared experiences; my partner, Alicia C. Lerner, for her unwaivering love and encouragement; and my son, Owen C. Lerner, who despite his best toddler efforts to prevent me from concentrating on my work has brought a new level of love and motivation to my life.

Contents

List of Figures

List of Tables

List of Acronyms

ACSL	ANSI/ISO C Specification Language
AXI	Advanced eXtensible Interface
CHARE	configurable hardware-assisted application rule enforcement
CORBA	Common Object Request Broker Architecture
CPS	cyber-physical system
CPC	cyber-physical controller
CVSS	Common Vulnerability Scoring System
DFSTAR	design-for-security, -trust, and -reliability
DREC	Datapath Rule Enforcement Controller
ECU	electronic control unit
FIFO	first in, first out
FPGA	field programmable gate array
FreeRTOS	Free Real Time Operating System
GPP	general-purpose processor
GTRI	Georgia Tech Research Institute
HDL	hardware description language
HLS	high-level synthesis
I/O	input/output
ICS	industrial control system
IT	information technology
MILS	Multiple Independent Levels of Security

MTU	master terminal unit
NGSCB	Next-Generation Secure Computing Base
NZ	normal zone
OS	operating system
PCS	process control system
PID	proportional-integral-derivative
PLC	programmable logic controller
RTU	remote terminal unit
SCADA	supervisory control and data acquisition
SerDes	serializer/deserializer
SGX	Software Guard Extensions
SoC	system-on-chip
SZ	secure zone
TAIGA	Trustworthy Autonomic Interface Guardian Architecture
TECEP	trust enhancement of critical embedded processes
TPM	Trusted Platform Module
TR	trust requirement

Chapter 1

Introduction

A cyber-physical controller (CPC) uses computational components to control a physical process. CPCs vary in size and capability, from microcontrollers in small-scale consumer products, to high-performance embedded devices in autonomous systems, such as field programmable gate arrays (FPGAs), and on to more complex devices in large-scale transportation and industrial control systems (ICSs), such as programmable logic controllers (PLCs). They include the subset of cyber-physical systems (CPSs) which control (as opposed to only monitor) physical processes, and process control systems (PCSs) which use a computational (as opposed to physical) controller mechanism.

CPC operations rely on trustworthy operation of embedded systems, which control physical processes directly. Unfortunately, trust is an increasingly difficult characteristic to sustain in modern embedded computing as their capabilities grow and tailored security solutions lag behind. Sophisticated cyber attacks like Stuxnet and Aurora have demonstrated that embedded components of CPCs are penetrable, even in air-gapped environments, and when compromised have a more devastating effect than previously observed in cyber security: physical destruction [2, 5]. The focus on second-generation approaches for embedded security adopted from general-purpose computing and over-reliance on network perimeter defense is failing to protect embedded controllers. More approaches to designing trusted leaf node components specifically are needed as it is increasingly clear that security for embedded systems is paramount in CPCs.

This work develops a root-of-trust design architecture, Trustworthy Autonomic Interface Guardian Architecture (TAIGA), which provides process resilience to cyber attacks. Our threat model includes malicious commands from supervisory units, network penetrations, denial-of-service attacks on controller interfaces or external components, Trojan controller code, surreptitious controller firmware updates, and communication compromises such as man-in-the-middle attacks on actuator and sensor links. The primary objective of TAIGA is autonomous preservation of physical process security and safety specifications regardless of the cyber-initiated threat. TAIGA is integrated with an embedded controller and assumes accurate system specifications can be enforced on its interfaces using hardware-implemented guards. A full implementation can predict and preempt specification violations, as well as vet firmware updates, assuming accurate models of the physical process can be integrated into the TAIGA root-of-trust. TAIGA protection logic accepts limited, range-bounded updates to specification guards values, but is otherwise static unless physical access is obtained. Physical attacks on cyber-components are outside of the scope of this protection scheme.

TAIGA was conceived from the confluence of several disciplines that are normally treated separately, including: control systems engineering, security engineering, model-based design, system-on-chip development, and source code formal analysis. We define five *trust requirements* for building a fine-grained trusted computing component. TAIGA satisfies all requirements and addresses all classes of CPC attacks using an approach distinguished by adding resilience to the embedded controller, rather than seeking to prevent attacks from ever reaching the controller. TAIGA provides an on-chip, digital, security version of classic mechanical interlocks. This last line of defense monitors all of the communications of a controller using configurable or external hardware that is inaccessible to the controller processor. The interface controller is synthesized from C code, formally analyzed, and permits run-time checked, authenticated updates to certain system parameters but not code. TAIGA overrides any controller actions that are inconsistent with system specifications, including prediction and preemption of latent malwares attempts to disrupt system stability and safety.

The following chapters provide a detailed overview of TAIGA and case studies of some of its applications. They also illustrate the exploratory evolution of our work. As different properties of TAIGA, applications, and the problem landscape emerged and were better defined, our terminology shifted to better describe these things. TAIGA, or components of TAIGA, are referred to with different terms in later chapters (earlier work), such as trust enhancement of critical embedded processes (TECEP), design-for-security, -trust, and -reliability (DFSTAR), configurable hardware-assisted application rule enforcement (CHARE), and Datapath Rule Enforcement Controller (DREC). Earlier work also often refers to CPCs as more general terms such as CPSs or PCSs. The remainder of this dissertation is organized as follows:

- Chapter 2 is a journal manuscript currently under review. Co-author attribution includes: Christopher J. McCarty, Kevin G. Lyn, and Cameron D. Patterson. This work defines CPCs and provides a survey of trust and security approaches for embedded systems. It also specifies five requirements for developing a fine-grained trusted component. TAIGA is described in terms of how it satisfies the requirements. An experimental application to a robot is also described as an example of applying TAIGA to autonomous systems.

- Chapter 3 is a manuscript presented at the 2014 International Symposium on Resilient Control Systems. Co-author attribution includes: Ron J. Prado, Zane R. Franklin, and Cameron D. Patterson. This work first fully defined TAIGA and studied it in the context of ICSs. A significant advancement from previous work was adding update functionality to TAIGA for specification guard values. TAIGA's hardware monitor functionality was also enhanced to average readings to be more resilient to CPC conditions. An additional form of formal analysis, value analysis, was also applied. Experimental results for a motor controller model are provided.

- Chapter 4 is a manuscript presented at the 2014 IEEE/IFIP International Conference on Dependable Systems and Networks. Co-author attribution includes: Zane R. Franklin, William T. Baumann, and Cameron D. Patterson. This work developed a full development and verification flow for TAIGA using high-level techniques. High-level synthesis for C code is used to create TAIGA hardware protections. The C code is formally analyzed using deductive reasoning with the Frama-C verification platform [3]. Experimental applications on a proportional-integral-derivative (PID) controller are provided. The full source code, including verification proof annotations, are provided in Appendix A.

- Chapter 5 is a manuscript presented at the 2012 International Conference on Security of Internet of Things. Co-author attribution includes: Mohammed M. Farag and Cameron D. Patterson. A major advancement of this work was the addition of prediction of controller malware or errors to TAIGA. It also showed that this prediction capability can be used to vet controller updates. Experiments for a aircraft pitch controller application are provided.

- Chapter 6 is a manuscript presented at the 2011 International Conference on Field Programmable Logic and Applications. Co-author attribution includes: Mohammed M. Farag and Cameron D. Patterson. This initial work first identified the utility of hardware-assisted application rule enforcement in a TAIGA-like approach. We also defined a high-level design flow for TAIGA and explored high-level abstractions for defining specification guards, such as SystemVerilog assertions or Bluespec SystemVerilog rules with guarded atomic actions. An example application in cognitive radio is presented.

- Finally, Chapter 7 discusses conclusions and future work. It also provides a listing of relevant publications, funded proposals, and invited talks resulting from this work as it pertains to the author of this dissertation.

Chapter 2

Trusting the Leaf Nodes: Embedded Security in Cyber-Physical Control

2.1 Abstract

General-purpose computing and network approaches to security are failing to protect cyber-physical control systems from cyber attacks targeting embedded controllers, which can have the direct effect of process disturbance or destruction. Moreover, as embedded systems increasingly grow in capability and find application in control systems, embedded leaf node security is gaining priority. We specify five requirements for building a fine-grained trusted computing component. We then provide a survey of recent approaches to security in embedded systems classified by their enforcement mechanisms. Lastly, we present a root-of-trust design architecture which satisfies all five trust requirements and provides process resilience to cyber attacks on, or from, embedded controllers.

2.2 Cyber-Physical Control

A CPC uses computing to control a physical process. CPCs are often referred to more generally as cyber-physical systems or process control systems, though those terms do not necessarily imply a *cyber* component *controlling* a process. Examples of modern CPCs can be found in self-driving automobiles, unmanned aerial vehicles, and other autonomous systems.

They are also used in large-scale ICSs, manufacturing, and utility infrastructure. Figure 2.1 illustrates a typical CPC architecture, including two classes of internal connectivity with respect to embedded controllers: sensor and actuator connections; and networking to supervisory, management, and human-machine interfaces. Supervisory units, and even sometimes embedded systems, are often connected to larger corporate networks or directly to the Internet to enable remote monitoring and control. The growing increase in networking is in part driven by the need for remote actuation and monitoring which reduces expenditures related to on-site services.

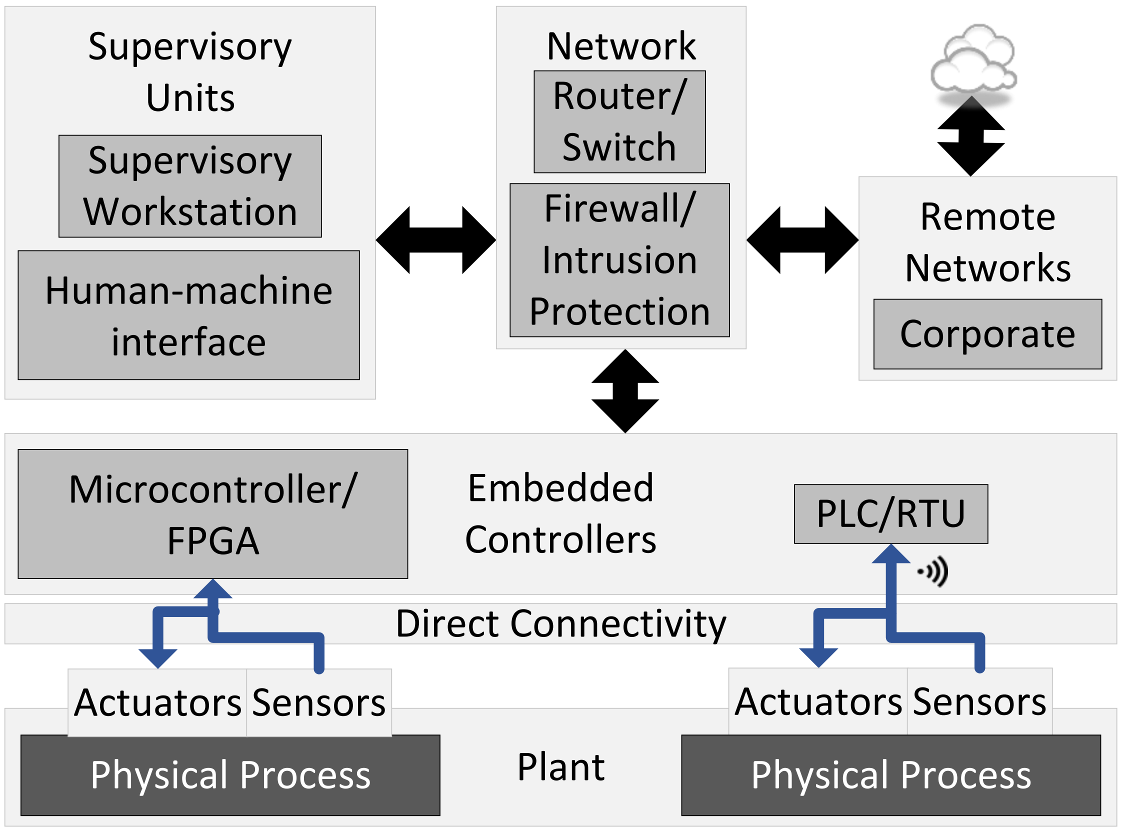


Figure 2.1: Generic cyber-physical control system

CPCs rely on embedded devices, sometimes referred to as field devices in this context, to control essential physical processes [23]. Examples of embedded devices for autonomous systems include single microelectronics such as general-purpose processors (GPPs) and FPGAs.

For instance, NASA's Mars rover Curiosity, which has closed-loop control systems for processes such as extraterrestrial drilling, relies primarily on FPGAs [8]. Examples for ICSs include PLCs, remote terminal units (RTUs), and programmable automation controllers. Embedded devices connect directly to actuators to control physical processes, and sensors to measure process state. They also connect to a network to report process status, and receive commands and configuration updates from supervisory control components. Embedded and supervisory systems are typically networked through traditional information technology (IT) infrastructure. Human-machine interfaces at supervisory units are used to monitor, control, and update processes and embedded components.

A complex CPC, such as an industrial supervisory control and data acquisition (SCADA) plant, often contains 100's to 1000's of control loops. The embedded controllers are usually networked and combined with master terminal units (MTUs), which are often just general-purpose computers running supervisory software. Control loops can be grouped together into subsystems and cannot always be considered independently. CPCs often contain numerous distinct and sometimes competing subsystems, such as a car's lane assistance and collision avoidance controllers.

2.2.1 Cyber Threats to CPCs

A large CPC can distribute trust across many computer nodes, communication links, and software layers within nodes. Many nodes are technologies developed initially for personal and IT platforms that eventually appear in CPCs. As a result, cyber threats are also migrating from computer systems used mostly for exchanging information and processing data to systems controlling physical processes. Similar to general-purpose computing platforms used mainly by people, CPCs often have the conflicting requirements of security and remote access. Preventing malware infiltration is difficult in complex, networked CPCs having zero-day exploits. Malware objectives are different for these two environments: interactive computing platform malware seeks information or computing resources, while sophisticated CPCs malware seeks to degrade or destroy the processes being controlled. Sources of disturbances can come in many forms, including insider threat, malware, malicious control commands, malicious controller updates, Trojans, or compromised sensor data.

CPC operations rely on embedded systems having real-time, high-assurance interactions with physical processes. Unfortunately, embedded systems are especially vulnerable to modern security threats and difficult to trust as they are often assembled from designer-generated hardware descriptions, software, and third-party cores. All of these sources can harbor undocumented, errant, and Trojan behaviors. Most code in ICSs is not even digitally signed. Ensuring trust in third-party modules individually is extremely challenging because there is neither an accompanying specification to trust in nor a golden version to compare to [27]. This problem is amplified and may result in system security violations when a embedded system is composed of numerous modules interfacing in a poorly trusted or understood manner. Moreover, the software tools used to design and implement these modules are themselves vulnerable to errors and insider threats. Thus, with the threats facing modern system development, it is reasonable to assume that some of these threats are built-in to the controllers themselves. A system in total can be considered untrusted and insecure until novel techniques are adopted to secure its most basic underlying components, e.g. embedded controllers [27].

Recent studies outlined potential cyber attack vectors for electronic control units (ECUs) in a large number of automobiles [19]. Researchers also demonstrated remote code execution on a telematics unit of a vehicle by exploiting the Bluetooth stack of an ECU [5]. There also continues to be an increasing number of incident reports received regarding actual cyber attacks on CPCs used in critical infrastructure. There is speculation that ICS espionage started as early as 1982 when a Siberian gas pipeline was sabotaged via implantation of controller malware eventually causing excessive gas pressure which lead to an explosion with roughly one-sixth the power of the first atomic bomb [22]. The Aurora attack demonstrated by the U.S. Department of Energy and Idaho National Labs destroys an electric generator by causing an embedded relay to intentionally open a breaker connecting the generator to the electric grid and then close the breaker after they shift out of synchronism, all within allowable tolerances so as not to activate protections [5]. A takeaway from Aurora is that CPCs must contend with not only accidental faults to the system but also targeted attacks seeking to damage equipment.

Perhaps now the most famous CPC attack, Stuxnet, highlighted embedded system vulnerabilities and the inadequacy of existing security solutions. It demonstrated that malware is not restricted to networks and general purpose computers, rather embedded systems used in CPCs can be infected as well. The goal of Stuxnet is to sabotage a specific physical system by reprogramming embedded controllers to operate outside their nominal bounds by intercepting routines that read, write, and locate PLC commands and data. Antivirus software missed the attack because PLC rootkits hide Stuxnet modifications to the system, and two stolen certificates validate new drivers. It also showed that air gaps and effective physical security measures are not a sufficient defense [2].

Cyber-space presents a new domain of warfare that is a great equalizer, unlike the other domains: land, sea, air, and space. In the event of large-scale hostilities, factories and infrastructure would likely be targeted by computer viruses. Smaller entities can now be major players as evident by the difficulty of identifying sources of attacks and the fact that newly identified zero-day exploits are even a commodity sold on brokerages to interested parties including nations. It is not clear how to discourage CPC attacks since anyone with Internet access can develop and deploy cyberweapons. It is clear, however, that current approaches to CPC security are lacking and more attention must be paid to developing and employing them.

2.2.2 CPC Cybersecurity

Similar to classical reliability engineering, the primary security objective for CPCs should be to bolster resilience to abnormal conditions. Traditionally, resilience for CPCs is defined as the ability to maintain stable operation in the presence of faults and disturbances. Many safety-critical systems, such as nuclear power plants, have self-contained mechanical interlocks to autonomously enforce system limits. However, interlocks are not widely used among most CPCs and typically only guard against catastrophic physical process states, acting as a fail-safe. Since such interlocks may halt process operation until reset, and thus not add to resilience, triggering them could indicate a successful security breach. Control-theoretic approaches are developed to bolster resilience by mitigating independent disturbances and faults. However, they are not developed to cope with cyber threats [28]. Resilience in the context of cyber components must consider a much broader set of conditions (or attack vectors) leading to disturbances, such as malicious supervisory commands, network intrusions, malicious firmware or code injections, man-in-the-middle communications compromises, and Trojan control logic.

As a result of utilizing IT infrastructure, security issues have carried over from IT systems to CPC systems. Networks, host operating systems (OSs), and users have traditionally been considered the least trusted elements of the traditional IT domain. Hence, most efforts of security researchers for CPCs have still focused on protecting networks and software from illegal access or malware propagation. Commonplace security measures in the IT domain that are still finding their way into CPCs include firewalls, intrusion detection systems, encrypted communications, user authentication, and access control. Unfortunately IT and personal device perimeter defenses are not a sufficient countermeasure since intrusion protection is initially reactive rather than preventative. As the Heartbleed exploit recently demonstrated, the dependence on complex protocols for secure communications will continue to be plagued by implementation flaws [25]. Most critically, traditional IT cyber security approaches do not consider the interdependencies between cyber and physical system components [28].

Higher-level protections in a large CPC – including network firewalls, MTUs, and even human operators – still rely on correct functioning of embedded systems at the leaf nodes. Yet, modern security approaches do not safeguard against tailored embedded system threats such as firmware or code modifications [23]. Protocols like industrial Ethernet are currently being adopted for embedded devices used in CPCs, exposing them to network-penetrating threats. This is worsened by the fact that embedded platforms also typically do not incorporate sophisticated network defenses. General-purpose computing device security, such as anti-virus software, is often reactive and not available for heterogeneous embedded systems. As a result of the dependence on network security and because embedded device security is lacking in this domain, there is relatively little for an attacker to do once a network is penetrated.

Recent attacks are evident of the need to mitigate CPC cyber threats not by bolstering perimeter security, but rather by assuming that potentially all layers of networks and software have already been compromised and are capable of launching a latent attack while reporting normal system status to human operators. To accomplish this requires a system security attribute that manifests quite differently in the IT and CPC domains: add safeguards as close as possible to what is being protected. For CPCs this suggests protections in the leaf nodes, i.e. embedded devices.

There is recent work on monitoring embedded devices in CPCs for security purposes. For instance, the WeaselBoard connects to a PLC backplane to forward intermodule communication to an external analysis system monitoring for anomalies [23]. Approaches to bolstering control resilience have also been researched, most of which have origins in fault detection techniques which are typically based on observing either physical process responses to new controller inputs or controller responses to new sensor inputs. A control protection architecture based on monitoring physical process responses to detect faults is described in [13]. The response of the physical process is monitored by decision logic that determines if a process violation has occurred. If a violation is detected, dedicated decision logic switches control to a high-assurance and presumably slower version of the controller until the system is stabilized. Unfortunately, system stability cannot always be recovered as the controller fault is not detected until after it has caused the physical process to deviate from allowed operational limits.

A protection architecture based on observing controller responses to new sensor inputs is presented in [6]. A process response is sent to both the regular high-performance version of the process controller and a trusted benchmark version of the controller algorithm. The responses of both controllers are used to generate a residual that can be evaluated to determine if a controller fault has occurred. Unfortunately, in this architecture the physical process is already being affected by the erroneous controller output by the time the controller fault is detected and corrective actions, such as switching over to a high-assurance version of the controller, can occur. This can result in the inability to return the system to a stable state before damage is incurred.

There has also been a recent progress toward designing tailored trustworthy spaces or roots-of-trust for embedded systems. The objective of trusted computing is to provide assurances that a system will dependably behave as expected, and do so without concealing additional functionality. For instance, CodeSeal tries to instill trust in an embedded device by using an external trust anchor implemented on a separate embedded device, such as an FPGA, to obfuscate and authenticate its code [23]. Yet in existing approaches, the main distinguishing feature of CPCs — a singular focus on regulating the state of physical processes — is not normally exploited. This is an important consideration for development of security solutions as even the most complex attacks, i.e. on par with Stuxnet, cannot disguise their ultimate goal of disturbing process stability. In the next section we examine more general, current approaches to embedded systems security, most of which have yet to be adopted in CPC applications.

2.3 Trust Enhancement of Critical Embedded Processes

Trusted computing protections for embedded systems must cope with a myriad of cyber threats. Some of the general types of attacks that should be considered when developing a trusted component are listed in Table 2.1. The objective of a cyber attack is typically to gain unauthorized access to, modify, or disable a system. The attacks described here are similar to those for general-purpose computing systems. Yet, the layered security protections typically used to address these threats in more general systems are typically lacking in embedded systems.

Table 2.1: Examples of cyber attacks on embedded processes

Attack Category	**Compromise of Trusted Component**
Malicious commands	Supervisory or other external commands which are not validated by the component may be used to maliciously bypass or co-opt the component
Network intrusions	Access to networked components may bypass higher-level IT or supervisory security measures and provide an ability to surreptitiously interact with or modify the component directly
Denial-of-service	The interfaces of the component might be removed or flooded in an attempt to cause the component to crash or be otherwise unable to performs its expected services
Unauthorized access and privilege escalation	Vulnerabilities enabling access to protected layers can compromise virtual separation of resources or be used to bypass protections altogether
Code injection and memory modification	Tricking the component to execute code or modify protected memory can be used to escalate privileges or operate the component outside of its intended use
Malicious updates	Modifications (including downgrades) to firmware versions can insert malicious functionality or disable critical security features
Trojan logic	Difficult to verify, closed (e.g. third-party), or unprotected source code may embed malicious triggers or actions into the component's innermost layers

In the context of CPCs, trust strategies for embedded components should also have the ultimate objective of ensuring physical process stability. Regardless of the cyber attack vector or cyber component corrupted in the overall system, a trustworthy CPC process should continue to achieve its objective of secure physical process control. Therefore, rather than analyzing a component's resilience against specific cyber attack types, assumptions, vectors, and countermeasures, the relevant analysis is whether a trusted CPC component both follows a scrutinized set of trusted computing requirements and enforces physical process stability specifications.

We propose five such *trust requirements (TRs)* which should be followed for establishing a trusted process architecture. A fine-grained trusted component has the following design-time and run-time attributes:

TR1 The source code and implementation for the entire component is analyzed.

TR2 The component uses private hardware resources for computation, internal communication, and memory, and does not invoke external components as subfunctions.

TR3 All external communication with untrusted components is through hardware-implemented, bounded, and isolated queues.

TR4 The component cannot be bypassed or disabled, and has a fixed repertoire of essential services, such as I/O or cryptography.

TR5 Critical functionalities of the component, such as rule checking logic, cannot be updated without provably secure or physical access.

Existing trust enhancement efforts can be broadly classified as efforts to establish secure design practices, efforts to validate and verify systems during development or pre-deployment, and efforts to build-in trusted features into a system that will be enforced post-deployment throughout the lifetime of a system. In the following section we examine approaches in each category including how well they satisfy the five TRs. In the next section we then propose an architecture which satisfies these TRs and preserves physical process security regardless of cyber attacks outside of the trusted component.

2.3.1 Secure Design Practices

The overall trust of an embedded system can be improved by applying traditional secure design practices, such as protected firmware updates with certificates, avoiding hard-coded critical values such as maintenance passwords, and using built-in authentication and encryption features. However, for embedded controllers more careful attention needs to be paid in the way designs are implemented. For instance, strict reliability or security driven partitioning and isolation of processes and resources is a requirement in building trusted designs. Reliance on OSs, hypervisors, virtual machines, or other multi-core security approaches to preserve isolation can be a dangerous practice as they all multiplex the same physical system resources including processors, memory, buses, and peripherals, thus violating TR2. Critical operations performed by a trusted module should be independent of any external modules or interfaces, even if the external module has greater privilege, and should have dedicated physical resources.

As TR2 implies, isolation considerations include not only resource sharing, but also trusted or untrusted component relationships and interfaces. McLean and Moore showed that fences, or reserved resources, and red/black analysis can be applied to certify a cryptographic solution on a single FPGA device [18]. Bus macros with direct routing are carefully inserted to allow communication between isolated regions. Huffmire et al. developed a system of moats and drawbridges to provide module isolation and communication control for multiple interacting cores [11]. Moats are similar in concept to fences in that they ensure modules do not share resources enabling communication between them. Drawbridges, which can be opened or closed, are then used to limit a module's ability to send and receive information on an interface, such as a connection to a shared bus. Drawbridges may also help to prevent modules from propagating the effects of undesired behaviors to each other.

Even a design decision made for cost, performance, or ease of implementation reasons can have security consequences. For instance, the basic choice between software or hardware as a target technology for critical functions. Consider the small sample of code in Figure 2.2. A software implementation would be susceptible to stack-corrupting buffer overflows, whereas a hardware implementation would neither be susceptible to memory corruption or code injection. A key strength of hardware over software implementation is the removed need for a contiguously addressable memory structure. Other simple examples of where hardware implementation would improve security, such as code with integer overflows or wraparounds, stored hard-coded credentials, or dangerous functions, are typically easy to identify even in code developed with good intentions.

```
void copy(char *arg) {
        char buf[10];
        strcpy(buf, arg); }
```

Figure 2.2: Code for a simple copy function

2.3.2 Pre-deployment Trust Approaches

Pre-deployment approaches seek to establish trust through evaluation or verification of hardware and software components. Hardware fabrication might be inspected in a sample of devices through a variety of destructive and non-destructive techniques, such as sophisticated imaging of chip layers [4, 14]. In an effort to satisfy TR1, designs can be scrutinized through static analysis and formal verification techniques, such as simulation, automatic test pattern generation, assertion verification, or model and information flow checking. Evaluation methods can also be applied on an implementation of a prototype or final system, such as functional testing or emulation with property checkers. Equivalence checking between various stages of the design implementation may even be used to ensure trust in the design tools themselves.

State-of-the-art trusted computing solutions are limited in their ability to provide assurances on control flows in embedded systems. Pre-deployment techniques generally focus on evaluating modules individually, especially when they are acquired from various sources. These techniques do not always provide the ability to achieve a given trust confidence level within a reasonable amount of time. For example, simulation and emulation techniques are sometimes ineffective in finding Trojans and illegitimate behaviors that are difficult to activate and observe. Verification of software modules is limited by the state-explosion problem. Verification of hardware is typically more achievable but can require detailed knowledge of the module in question. A common limitation is that most evaluation methodologies require the existence of golden references [27].

Perhaps the greatest limitation of pre-deployment trust techniques is the reliance on access to source code. Unfortunately, a great deal of the code used in CPCs is only available as binary code, which limits what can be verified with source code analysis or compilers. Embedded systems are far less amenable to inspection than general purpose computers due to insufficient examination (e.g. reverse engineering) tools and suitable interfaces [23]. Access to running systems may not always be feasible making verification techniques relying on dynamic analysis unpractical.

2.3.3 Post-deployment Trust Approaches

The constant race to patch system vulnerabilities against newly discovered exploits and the difficulty in constructing systems which can be fully verified are indicative of the need for proactive protections to be built-in. Post-deployment solutions focus on building trust into the system using dedicated resources that provide critical security functions or evaluate security aspects at run-time. While such protections may be difficult to provide for highly complicated, general-purpose desktop and server computing platforms, it is possible to create them for embedded systems intended for specific applications. Post-deployment trust technologies used in general-purpose and later embedded systems can be classified by their enforcement mechanisms. Protections are built from trusted software or hardware components which limit untrusted processes or access to resources. Some approaches use a combination of both software- and hardware-limiting schemes to distribute trust responsibilities to meet security or performance requirements.

Software-limiting schemes establish a hierarchy of trust between software layers, the innermost of which are given exclusive responsibility for allocating and managing hardware resources such as GPPs, memory, and peripherals. Trusted software processes are used to vet requests from less trusted processes or users. Examples technologies include software firewalls, virtual machines, and hypervisors. Multiple Independent Levels of Security (MILS), for instance, uses a trusted kernel base-layer to maintain process and data separation between partitions for applications [2]. The application in each partition runs on top of a MILS middleware service-layer that provides the mechanisms to enforce security. The architecture can use a scrutinized version of real-time Common Object Request Broker Architecture (CORBA) to communicate between partitions.

Another example in the embedded computing domain is ARM's TrustZone technology which segments hardware and software resources based on levels of trust [1]. TrustZone assigns applications and resources to either non-secure or secure partitions. Applications from both partitions are executed on shared processing resources, using either a trusted or non-trusted OS. A security monitor is used to switch between secure and non-secure kernels as needed as needed by the applications, and those operating in the non-secure OS are restricted from accessing trusted peripherals and memory addresses. An insecure design attribute of this approach is resource sharing which violates TR2.

A security concern with software-limiting solutions is that they depend on software correctness and its ability to resist malware. Software's verification complexity makes it susceptible to known vulnerabilities, such as using buffer overflows to execute code that masquerades as data, and the possibility for zero-day exploits. TrustZone and MILS have grown in code complexity to the point that TR1 can no longer be easily satisfied in most implementations. A recent TrustZone vulnerability was discovered in which an integer overflow flaw in Secure Monitor Call requests allows an attacker with escalated kernel-level privileges to issue controlled data writes to arbitrary secure memory, which could be exploited to execute arbitrary code [21]. Along these lines, the only potential security advantage over an OS that a hypervisor might have is that it is simpler. However, a production hypervisor will still likely not be amenable to formal validation. Additionally, software-oriented solutions do not always offer the performance necessary to monitor high-speed, real-time, embedded systems.

In hardware-limiting schemes, typically static hardware units are used to separate software processes and data. Hardware controllers can also have exclusive responsibility for managing hardware-implemented processes or channels, and can deny requests from software at any layer. Examples for general-purpose computers include hardware firewalls, the protected input and graphics components of Intel's Trusted Execution Technology [12], Intel's upcoming Software Guard Extensions (SGX) [13], and the Trusted Computing Group's Trusted Platform Module (TPM).

A TPM, whose main function is to preserve platform integrity, provides remote attestation by creating hash-key summaries of system configurations as well as cryptographic functions for secured data binding and storage [10]. Standalone TPMs satisfy all five TRs, though they are typically used as sub-modules to software (thus violating TR2 and potentially TR4), not to handle critical interactions with physical processes. Microsoft's Next-Generation Secure Computing Base (NGSCB) is a software architecture built on the security provided by a TPM [19]. The TPM is primarily used to ensure that the system starts in a known good state. NGSCB consists of both trusted and untrusted mode kernels, with the trusted kernel providing a secured environment for trusted code to execute in. In the current architecture, applications must be written to take advantage of the security features provided. This approach also suffers to satisfy TR1. In an effort to satisfy TR3, a manager is built into the operating system to exchange data between the trusted and untrusted modes. In Intel's SGX, trusted execution environments, or enclaves, are created for applications in the context of potentially untrusted OSs, without the dependence on external TPMs.

An increasingly attractive hardware-limiting approach for embedded devices is to implement

TPM-like, run-time, hardware-based monitors that are derived from a system's operational or security specifications. Assertions or test benches used to test the system during the design phase can be translated into properties to check for in real-time. This design-for-trust protection scheme does not assume the existence of a golden reference or the ability to inspect the internals of third-party intellectual property modules [4]. It provides assurances for suspected behaviors that are difficult to activate or prove. Tailored, hardware-based checkers can easily satisfy most requirements and provide the performance necessary to deter high-speed attacks.

An example is the specialized language developed by Huffmire et al. to specify legal memory access policies for FPGA-based embedded systems [11]. The policies are synthesized into a reconfigurable hardware reference module that decides the legality of every memory access request generated from a datapath module. This work was further developed into a method of generating hardware-based security checkers to detect processor malicious inclusions at run-time [4]. Security-relevant invariants of a processor's architectural specification are described on corresponding circuits of the processor's design using Property Specification Language assertions. Security checkers are then automatically generated as synthesizable hardware to verify linear temporal logic properties of expected behaviors. A drawback to this approach to property specification is that it requires low-level knowledge of the designed circuits. This limits the ability for more abstract, specification-level security assertions to be made without incurring additional designer translations. The use of domain-specific and specialized languages also have drawbacks of unfamiliarity to designers, and limited ability to concisely capture complex monitoring and enforcement circuit behaviors.

For embedded devices Abramovici and Bradley proposed an application-dependent defense infrastructure in which datapath signals are monitored for unexpected or illegal behaviors [1]. The signals are selected by a designer directly in a hardware design language and grouped to create multiplexed probe networks that source information to security monitors. The hardware-based security monitors are configurable finite state machines that check the current set of selected signals for behavior properties also specified by the designer. The signal probe network and monitor configurations are controlled by a security processor, which may initiate countermeasures to failing security monitors.

Abramovici and Bradley assert that security checkers should be invisible to embedded software to increase security, which is consistent with TR2. Though, in their proposed protection scheme embedded software is used to control the hardware configurations of the distributed defense infrastructure. This approach has the advantage of being able to reason about complex properties of overall system performance or health testing. However, it is susceptible to software vulnerabilities and is unnecessarily complex if trust and security can be more tightly associated with suspicious modules. The system of countermeasures is also software-initiated, which limits performance and is often inappropriate for attacks executing at hardware speeds.

Unfortunately hardware-limiting advances from high performance desktop and server computing platforms are being adopted by embedded systems at a considerably slower rate than their software-limiting counterparts. For instance, hardware-limiting protections are typically not provided for control flow assurances in platforms that make up the majority of embedded processing. This adoption typically lags due to concerns that security measures will overwhelm tight cost and power constraints. When assurances are provided, they are typically aimed at protecting against physical attacks. Adoption is further complicated by the general-purpose nature of devices and intellectual property that embedded systems are built from.

For CPCs trusted computing architectures must also consider appropriate countermeasures to remain resilient against attacks. The Huffmire et al. monitoring-oriented solution does not discuss integration of countermeasures or real-time enforcement within a system. The authors do, however, indicate that a limitation of this approach is its susceptibility to attacks that are effective before they are detected and an opportunity for countermeasures is provided. In the scheme presented by Abramovici and Bradley, countermeasures are implemented by controlling specified datapath signals. When a security violation is detected, the software control processor may override signals or take actions to isolate a core. However, the authors acknowledge that broader countermeasures are required to create a system-level protection scheme. In general, it is likely an intractable problem to create real-time countermeasures tailored specifically to every possible attack on every system interface. Thus, a security designer must also be given the flexibility to create abstract countermeasures or real-time enforcement practices that align with system specifications. Table 2.2 provides a summary of trust requirement satisfiability for the security architectures discussed in this section.[1]

[1]Trust requirement satisfiability may vary between implementations

Table 2.2: Summary of trust requirement satisfiability for run-time protections

Trust Architecture	**TR1**	**TR2**	**TR3**	**TR4**	**TR5**
Hypervisor	**X**	**X**	**X**	✓	**X**
MILS	**X**	**X**	**X**	✓	**X**
TrustZone	**X**	**X**	**X**	✓	**X**
TPM	✓	✓	✓	✓	✓
NGSCB	**X**	**X**	✓	**X**	**X**
Tailored HW Checkers	✓	✓	✓	✓	✓

2.4 Trustworthy Autonomic Interface Guardian Architecture

The TAIGA addresses all previously described classes of CPC attacks using an approach distinguished by adding resilience to the embedded controller, rather than seeking to prevent attacks from ever reaching the controller. TAIGA provides an on-chip, digital, security version of classic mechanical interlocks. This last line of defense monitors all of the communications of a controller using configurable system-on-chip (SoC) or external hardware that is inaccessible to the controller processor. TAIGA overrides any controller actions that are inconsistent with system specifications, including prediction and preemption of latent malwares attempts to disrupt system stability and safety.

TAIGA is a trusted monitor which can both limit the external consequences of internal malware and detect external attempts to use an interface in an unsanctioned manner. TAIGA also serves as a trust anchor to check the integrity of updates, and is trustworthy even if application and task management software are not [15]. Unlike mode-based operating systems, hypervisors, or security extensions such as ARM's TrustZone, TAIGA adheres to TR2 by using true resource isolation rather than software-mediated context switching of shared hardware resources. Source code is not needed for application and supervisory code, and the only inputs to TAIGA are the CPC specifications and models.

TAIGA was conceived from the confluence of several disciplines that are normally treated separately, including: control systems engineering, security engineering, model-based design, system-on-chip development, and source code formal analysis. In order to enhance trust in critical embedded processes, TAIGA redistributes responsibilities and authorities between a controller and a configurable, hardware-implemented interface controller, simplifying controller software without degrading performance while separating trusted components from updatable software. The interface controller is synthesized from C code, formally analyzed to satisfy TR1, and permits run-time checked, authenticated updates to certain system parameters but not guard code. TAIGA's main focus is ensuring process stability even if this requires overriding commands from the controller or supervisory nodes. Adding trusted, application-tailored, software-inaccessible, autonomic hardware leverages commercial programmable SoC platforms without degrading real-time performance or increasing size, weight, and power, which is especially important for autonomous systems. By being independent of the embedded systems application, network, or OS software layers, TAIGA greatly reduces the attack surface and possible zero-day exploits. TAIGA is not domain-specific and can be developed for CPCs in many scenarios, such as unmanned systems, manufacturing, industry, and transportation.

Figure 2.3 illustrates how TAIGA is the sole means by which a controller's GPP communicates through any external interface to sensors, actuators, and networks. TAIGA uses hardware-implemented first in, first out (FIFO) modules for intermodule communication, thus satisfying TR3. All external traffic is monitored by the TAIGA, which can override GPP actions if a physical process is deviating from specifications. To mitigate a Stuxnet-like attack, the TAIGA can take corrective action (such as reverting to a backup controller) and generate alarms without GPP assistance should a controlled physical process risk instability due to malware-infected GPP code, malicious requests from supervisory units, or sensor malfunctions. Hence TAIGA satisfies TR4 acting as a physical process guardian by enforcing formally validated safety and liveness properties that are independent of GPP software actions or inactions.

TAIGA specification guards can be classified as operational or safety guards. Operational guards are enforced to meet the standard process specifications. They are typically simple, such as hard bounds on operating limits for the sensor measurements and control signals. Safety guards ensure that the process remains stable and protects the plant from safety hazards. For safety guard violations, defensive mechanisms such as relief valves and auxiliary power for electro-mechanical systems brings the process back to safety. Guards are isolated in hardware logic and can only be updated by physical access in the most secure scenario, thus satisfying TR5. However, to allow for flexibility in control algorithm or process tuning (such as coping with component aging), TAIGA does validate and allow range-restricted updates to particular guard enforcement values.

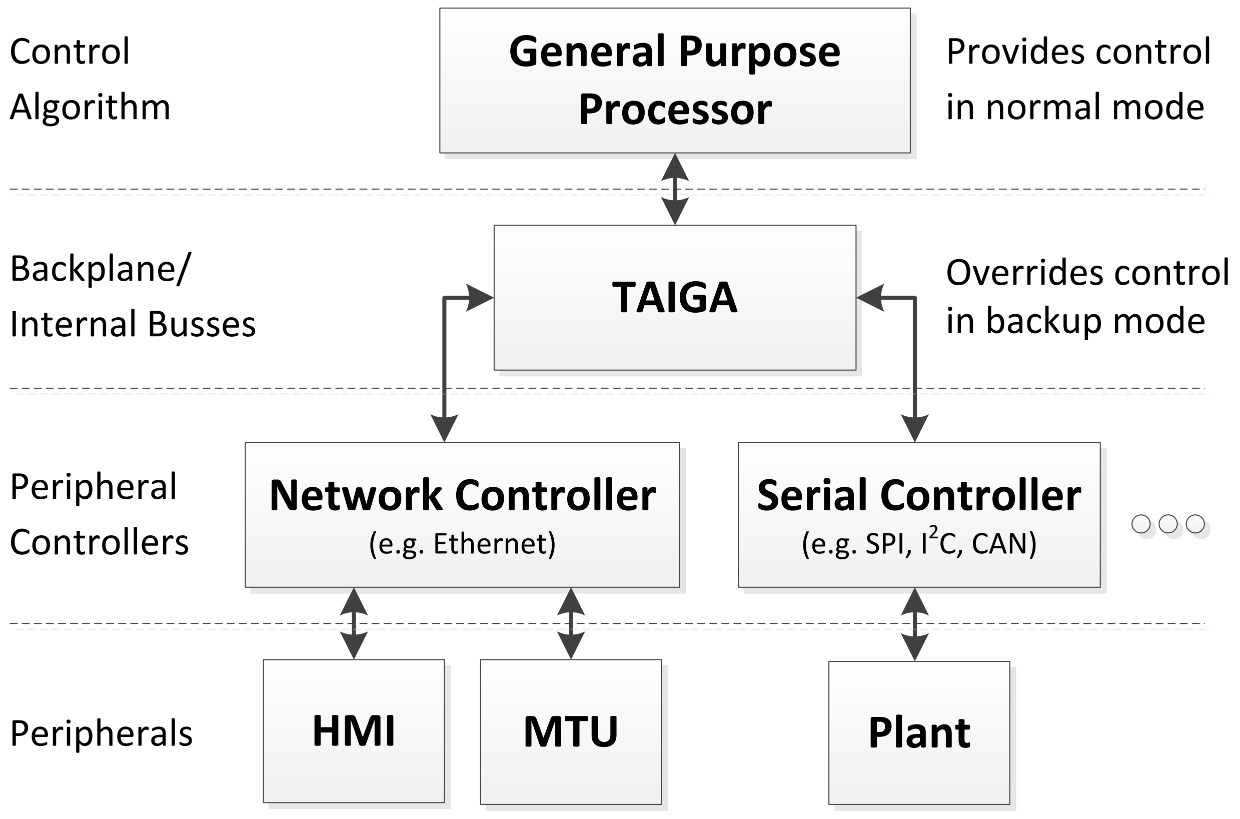

Control Algorithm
General Purpose Processor
Provides control in normal mode
Backplane/ Internal Busses
TAIGA
Overrides control in backup mode
Peripheral Controllers
Network Controller (e.g. Ethernet)
Serial Controller (e.g. SPI, I2C, CAN)
Peripherals
HMI
MTU
Plant

Figure 2.3: Abstract TAIGA architecture

TAIGA is capable of predicting malware attacks rather than just reacting to them. A copy of the controller application can communicate with an embedded model of the physical system. This virtual control loop can be accelerated to foresee latent malware in the production control code [7, 15]. If this occurs, the backup controller or other safety measure can be invoked preemptively. The model, specification guards, switches, and backup controller can all be implemented in configurable logic and therefore inaccessible to the GPP and network.

2.4.1 Illustrative Example: Robotic Control

This demonstration simulates a hazardous cargo-carrying mobile robot similar to those being adopted in industrial settings today. In high-risk environments such as these, physical protection and availability of plant equipment are critical. The demonstration platform is the commercially available Lynxmotion Johnny 5 robot, which utilizes differential drive treads as well as a small servo-controlled humanoid body. Two sets of downward facing reflectivity sensors allow the robot to perform line-following. C-based controller code runs on an ARM processor which receives commands via Ethernet from the plant's control network. Network commands may include operator or automated plant generated commands.

The robot follows a set of black and green floor markings as it uses its arms to grab and transport cargo between two drop stations. Because the robot performs a series of complex actions in a sequence, it utilizes different sets of control laws in each step. The set of control laws, which determines vehicle direction, line-following algorithm, and arm positioning, is determined by the value of a status variable. The status variable follows a state machine that describes the robot's behavior with respect to the overall process as shown in Table 2.3. The status variable may be incremented by a floor marking color transition, which indicates the robot has moved into a new physical zone, or by a timer, which indicates a specific action is complete.

Table 2.3: Robot status and control actions for cargo transfer application

Status		**Control Law**		
Id	**Overall Action**	**Tracking Algorithm**	**Hand Position**	**Motor Direction**
0	Wait for start	None	Dont care	Stopped
1	Grab at station 1	Forward	Dont care	Forward
2	Grab at station 1	Forward	Dont care	Forward
3	Grab at station 1	None	Ready	Stopped
4	Grab at station 1	Forward	Ready	Forward
5	Grab at station 1	None	Grabbing	Stopped
6	Drop at station 2	Reverse	Holding	Reverse
7	Drop at station 2	Reverse	Holding	Reverse
8	Drop at station 2	None	Dropping	Stopped
9	Wait for start	Reverse	Locked up	Reverse
10	Wait for start	None	Ready	Stopped
11	Grab at station 2	None	Ready	Forward
12	Grab at station 2	Forward	Ready	Forward
13	Grab at station 2	None	Grabbing	Stopped
14	Drop at station 1	Forward	Holding	Forward
15	Drop at station 1	None	Dropping	Stopped
16	Wait for start	Reverse	Locked up	Reverse
17	Wait for start	Reverse	Dont care	Reverse
18	Wait for start	Reverse	Dont care	Reverse

To simulate a supervisory control network, remote stations are permitted to establish shell connections to the control board. This enables control stations to update the controller code directly on the robot. Alternatively, a station can launch a separate run-time command executable which sends entered commands to a pre-determined file on the robot. In each iteration of the main control loop, the controller searches and parses this file, and immediately executes any recognized commands. Commands can include higher-level motor control actions, or lower-level modifications to controller variables and settings.

Vulnerabilities in the user interface, other plant components, or the network itself may prove to be easy attack vectors. Attacks might aim to damage or destroy the plant by directly commanding the robot or updating the controller code to drop its cargo and damage the plant. The Common Vulnerability Scoring System (CVSS) created by the Department of Homeland Security provides a method to quantify possible vulnerabilities and their potential damages on a scale from 0-10 [20]. Because moderately simple attacks on the control network can do catastrophic losses to the plant, the system contains multiple vulnerabilities with a CVSS score of 7 or higher, classifying the system as "high" risk. The CVSS v2 vector, which summarizes the reasoning for the score using a series of abbreviations, is `AV:A/AC:L/Au:S/C:C/I:C/A:C`.

TAIGA Functionality

By directly evaluating the controller commands, TAIGA's design allows it to prevent physical process damage regardless of the attack vector. In order to distinguish between normal and abnormal operation, TAIGA uses status information. As TAIGA sends sensor data to the primary controller, it also analyzes the readings to maintain its own status state machine very similar to the primary controller. It stores the value of the status in a state register shared with the backup controller, which enables that controller to take over correctly upon primary controller failure.

Each TAIGA iteration begins with a check to see if a command has been received from the primary controller, as shown in Figure 2.4. If a command has been received, TAIGA first performs a series of logical checks to ensure that the command is in accordance with the specification guards. The specification guards are created by the control designer as a tailored set of conditions which should be upheld, such as keeping motors within a safe operating speed. The specification guards are self-contained and application-specific, thus satisfying TR4.

Next, TAIGA verifies the command meets the operational guards, which vary depending on the status variable. Design of the operational guards begins with the control designer listing system stability states or possible actions which would result in system damage. The designer then looks at what commands or sequence of commands would lead to such damage from the each status, and blocks them in the operational guards. The operation guards can be described a type of a control law that, instead of describing which actions to take, describes which actions not to take. If the commands fail to meet either the specification guards or operational guards, TAIGA reverts to the backup controller.

If no command is initially received from the primary controller, TAIGA checks to see if a safety-critical command should have been sent in the current status. Safety-critical commands are defined by the control designer as high-priority commands such as stopping, properly dropping cargo, or performing line following. If an expected critical command does not arrive in a specified amount of time, TAIGA assumes the primary control has failed and reverts to the backup controller. In this experiment, the backup controller returns the cargo to its original station, and resets the robot to its starting location.

Implementation

Our robot system consists of two commercially available, Zynq-7000 SoC FPGA-based, ZYBO development boards. One SoC contains the physical process guardian, while the other contains the physical process controller. To satisfy TR3, the only point of contact between them is a dual 32-bit FIFO interface allowing for the bidirectional exchange of commands and responses. In separating the updateable and untrusted control software from the trusted physical process drivers and utilities, possible attack vectors through this FIFO interface are reduced. Standard hardware description languages (HDLs) are used for some aspects of implementation, as well as higher-level programming languages which are synthesized to hardware using Xilinx's Vivado high-level synthesis (HLS).

The physical process controller, outlined in red in Figure 2.5, runs a basic Linux distribution on ARM core 0. Note that the ARM has a solid border, indicating that is a hard core, as opposed to a dashed border which indicate soft cores. The programmable logic contains a dual 32-bit FIFO interface, as well as a 32-bit serializer/deserializer (SerDes) for communication with the physical process guardian. Linux handles all networking and scheduling,

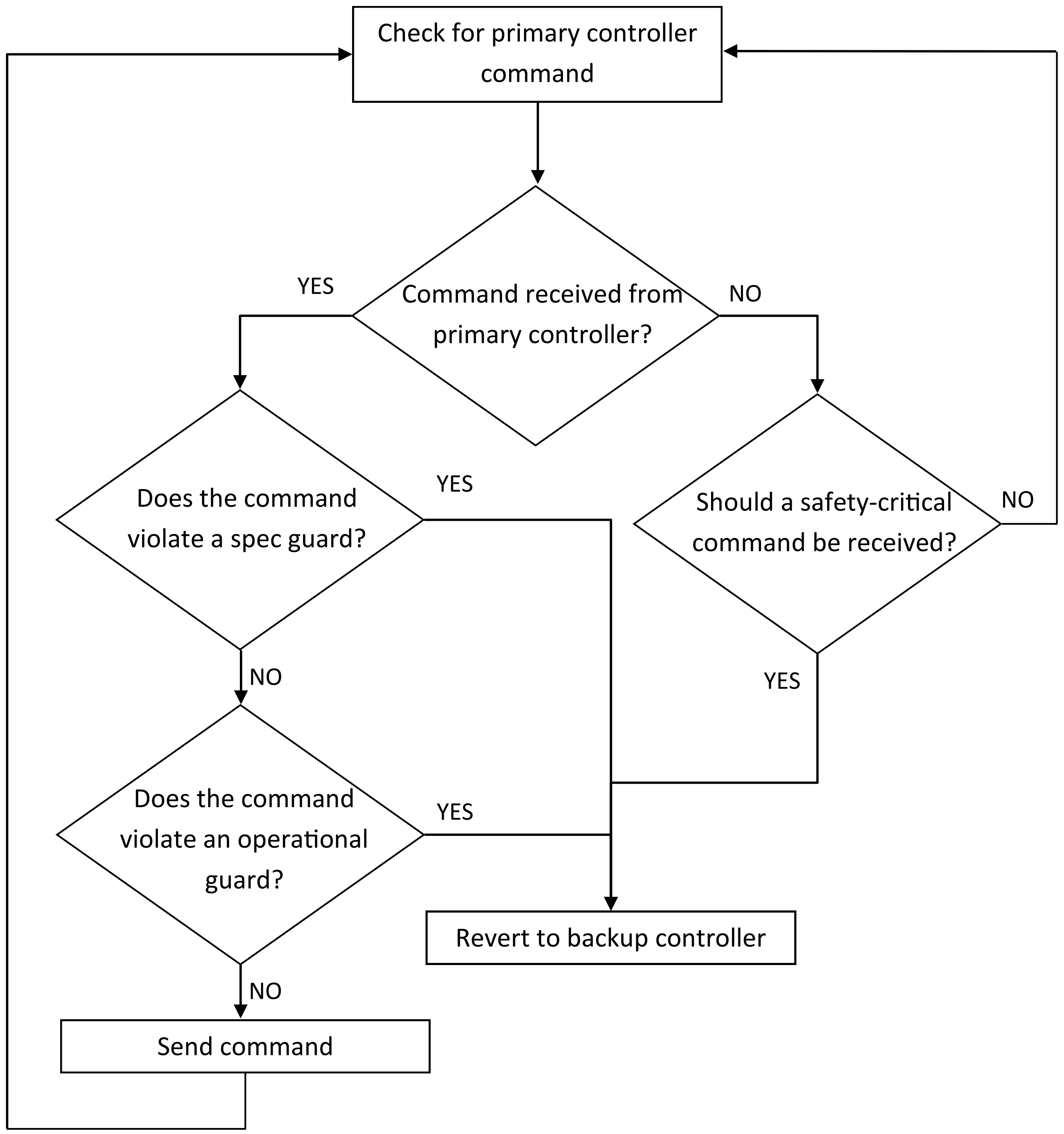

Check for primary controller command
Command received from primary controller?
YES
NO
Does the command violate a spec guard?
YES
Should a safety-critical command be received?
NO
NO
YES
Does the command violate an operational guard?
YES
Revert to backup controller
NO
Send command

Figure 2.4: TAIGA control flow

and ensures the physical process control application(s) are running. The interfaces to these FIFOs consists of a simple HLS module. The FIFO interface connects to the custom 32-bit SerDes which handles the 32-bit full duplex communication between the two separate SoCs. The physical process guardian is outlined in green. Commands from the process controller are transmitted to the guardian and vice versa.

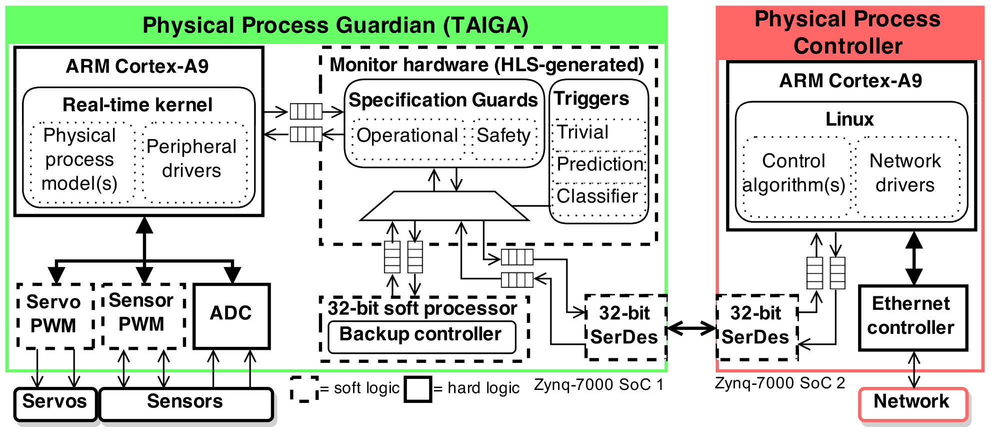


Figure 2.5: Process controller with TAIGA implementation using dual SoC devices

The process guardian runs Free Real Time Operating System (FreeRTOS) on ARM core 0. FreeRTOS maintains a model of the physical process, and makes the current state of the process known to the monitor via a simple four-bit process state register. It also maintains ongoing low-level control of the physical process, such as scheduled sensor reads or servo motion. Higher-level commands are received from the process controller SoC and executed, being already validated by the TAIGA. Some commands require responses containing sensor data, which FreeRTOS writes to the outgoing FIFO in the interface.

The TAIGA monitor is also implemented using HLS, and acts as an interface guardian. The internal modules within the monitor are parts of the C application used to synthesize the monitor hardware. Triggers determine which FIFO interface to use as the control source based on the physical process state and incoming commands from the process controller. When a new 32-bit command can be read from the FIFO interface, the monitor does so and enforces the specification guards. Some guards require contextual information about the physical process to be enforced, which is read from the four-bit state register maintained by the process model in the real-time kernel. The monitor considers a 32-bit command that does not violate any specification guard valid and writes it to the FIFO interface for execution by

FreeRTOS. Command responses from FreeRTOS follow a parallel control path in the opposite direction back to the physical process controller. The backup controller is implemented in a MicroBlaze soft processor running a bare metal application in the programmable logic. If triggered, the backup controller wakes, reads the current state of the physical process from the four-bit state register, and then stabilizes the physical process using a verifiable algorithm.

TAIGA protections are implemented entirely in isolated programmable logic, thus satisfying TR4, and in adherence with TR5 are static outside of the range-checked specification guard values. The logic configuration ports are not interfaced with any other processes in the system. To keep TAIGA flexible yet secured, physical access to logic is required to perform an update. Standard programmable security mechanisms, such as bitstream encryption and authentication, are used to protect process information and require maintenance personnel to have the appropriate credentials.

The additional hardware resources used are small percentages of the totals available on the two SoCs. As seen in Table 2.4, the highest utilized resource is the LUT at 9%. The type and size of FIFO used in the TAIGA interfaces is dependent on the platform and application, not fixed. The HLS modules used to accomplish this each use 206 FFs and 291 LUTs, which are only hundredths of a percent of the totals available. The monitor hardware, arguably the most critical aspect of the TAIGA, uses very few resources.

Table 2.4: TAIGA FPGA resource utilization

Component	**I/O**	**FF**	**LUT**	**BRAM**	**SRL16E**
Guardian Monitor	0	131	247	0	0
SerDes	8	316	168	0	0
Backup controller	0	980	1256	2	225
Total Available	100	35200	17600	60	6000
Percent Utilization	8	4	9	3	4

If it is not practical to add a separate chip to implement TAIGA, isolation of trusted and untrusted regions can also be accomplished with a single SoC by instantiating an additional MicroBlaze soft processor to run the real-time kernel. In this example application, a single SoC implementation would be quite similar to the dual SoC implementation. The two SerDes modules are no longer required. However, the real-time kernel application is larger than that of the backup controller, requiring a larger instruction memory for its soft processor. This results in additional required block memories, though much less than the quantity available.

The process guardian uses four different clocks in total, as shown in Tables 2.5 and 2.6. A 100 MHz clock runs process-specific cores which require that specific frequency. A 150 MHz local bus clock runs the Advanced eXtensible Interface (AXI) infrastructure, monitor hardware, and other peripheral cores. The SerDes link utilizes the external clock from the incoming serial channel as well as a local clock for the outgoing serial channel. These clocks are not synchronized, and run at either 50 MHz single-ended, or 200 MHz differential. The component which adds the most latency is the SerDes link, requiring 70 clock cycles to read a value from a FIFO, transmit it, and write it to a FIFO on a different SoC. A unified implementation on only one SoC would gain a significant drop in latency with a modest gain in hardware resource utilization. The latency added by the monitor hardware depends on the guards enforced.

Table 2.5: TAIGA latency analysis

Component Latency		
Component	**Clock Frequency**	**Additional Clock Cycles**
	MHz	**(Min : Max)**
Monitor	150	2 : 7
32-bit SerDes	50 or 200	35 : 35

Table 2.6: TAIGA timing path analysis

Timing Path Latency	
Path	**Worst Case Added Latency (ns)**
Linux to real-time kernel @ 50 MHz	1407
Linux to real-time kernel @ 200 MHz	38
Backup controller to real-time kernel	47

Formal Verification and Testing

Rigorous functional verification can be the most time consuming part of system development. However, we demonstrate that a HLS design flow enables the leveraging of high-level formal tools to expedite the verification process. Proofs are written in order to test properties directly on the software source code design for the TAIGA-augmented controller. Hence, the verification model does not require abstraction of the implemented source code. Proofs on the

C code are hardware independent and are not concerned with low-level timing. Additionally, in order to prove properties of security and promote TR1, the verification space is reduced from the entire system to only TAIGA additions.

For security analysis we are most concerned with evaluating TAIGA additions. The source code for a specification guard module annotated with ANSI/ISO C Specification Language (ACSL) is shown in Figure 2.6. In this example we use multiple function contracts, `verify_all_valid` and `verify_any_invalid`, to perform a case analysis of all possible values that could be passed to the specification guard. An `assumes` clause is used in each proof contract to provide variable ranges and make the function return value determinate in all conditions. The first contract, `verify_all_valid`, ensures that when the variables passed to the guard are within specification the function will evaluate to true (one). Similarly, the second contract, `verify_any_invalid`, ensures that when a variable is out of specification the function will evaluate to false (zero).

Frama-C's Jessie plugin was used to verify the proofs for the specification guard [3]. Every obligation was discharged by the Alt-Ergo prover, thus ensuring our confidence in the operation of the module. When multiple function contracts are used we are also able to reason about the overall set of behaviors. The `disjoint behaviors` and `complete behaviors` clauses indicate that the set of behaviors are complete and disjoint and should all be taken into account during analysis. The Alt-Ergo prover was able to discharge both of these checks ensuring all possible variable values were covered by the proofs. The provers were also able to discharge additional safety checks, such as array bounds checking. However, these safety checks are not as relevant to TAIGA since it is implemented in hardware.

Another useful Frama-C plug-in is *value* analysis which automatically computes variation domains for the variables of the program [9]. In other words, value analysis provides sets of possible values for the program variables. Value analysis is context- and path-sensitive, and generally considered most useful for embedded code. Aside from providing final variable value ranges for terminating functions or at various points throughout program execution, the plug-in also computes the truth value of any precondition, postcondition, or other user assertion as it is encountered. TAIGA supports limited updates to specification guard variables to cope with control algorithm adjustments or process tuning. Value analysis is used to ensure that these updateable parameters can never be set outside of specified limits [10].

The motor controller interface in our example application emulates a servo, allowing us to

```
typedef unsigned u8;
typedef unsigned u32;
enum States {DROPPED, GRABBED};
#define SERVO_POS_MAX 2500
#define SERVO_POS_MIN 500
#define validateThreshold(value, upper, lower) ((value >= lower &&
    value <= upper) ? 1 : 0)
// Function for operational guard on the servo position value
/*@assigns \result;
behavior verify_all_valid:
  assumes ((process_state == DROPPED) && ((command <= SERVO_POS_MAX) &&
      (command >= 1200))) ||
          ((process_state == GRABBED) && ((command <= 750) && (command
             >= SERVO_POS_MIN)));
  ensures \result == 1;
behavior verify_any_invalid:
  assumes ((process_state == DROPPED) && ((command > SERVO_POS_MAX) ||
     (command < 1200))) ||
          ((process_state == GRABBED) && ((command > 750) || (command <
              SERVO_POS_MIN)));
  ensures \result == 0;
disjoint behaviors verify_all_valid, verify_any_invalid;
complete behaviors verify_all_valid, verify_any_invalid;*/
u8 servoPositionOperationalGuard(u32 command, enum States process_state
    ){
        u8 retval = 0;
        if(process_state == DROPPED){
                retval = validateThreshold(command,SERVO_POS_MAX,1200)
                   ;}
        else if(process_state == GRABBED){
                retval = validateThreshold(command,750,SERVO_POS_MIN);}
        return retval;}
```

Figure 2.6: TAIGA specification guard function annotated with Frama-C

consider the servo position as positive and negative velocity. The graphs shown in Figure 2.7 display the positions of servos during different events of a physical test experiment. The gray areas indicate a range of acceptable positions for the given state, while a black line represents a single valid position for the servo at that particular instant. A dashed red line represents a failed attempt to set the servo to the indicated position, while a solid red line would indicate a successful manual override (if TAIGA had not prevented them all). These manual override attempts simulate malicious commands injected during actual run-time of the robot.

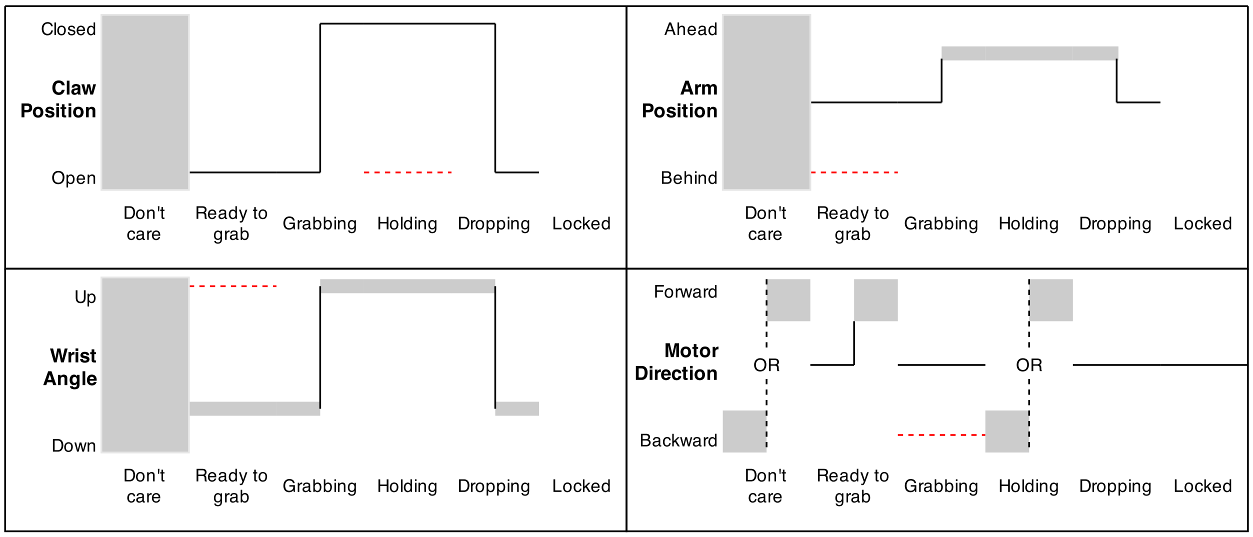


Figure 2.7: Robot with TAIGA experimental testing results

2.5 Conclusion

In this work we surveyed cyber threats to CPCs to illustrate that perimeter and network defenses to security are failing to protect against complex attacks on CPCs. Embedded controllers are increasingly penetrated in cyber attacks leading to disruptions in process operations and even physical damage. Unfortunately, tailored security at the CPC leaf nodes is lacking. We provided a taxonomy of existing approaches to embedded systems security, some of which have already been applied to aid in protecting CPC processes. We proposed five requirements that should be followed when creating a fine-grained trusted component and described how well existing security approaches adhere to them. We also proposed an architecture for embedded systems used in CPCs which satisfies all five of these requirements. TAIGA is an isolated, hardware-implemented, verifiable, tailored trustworthy space which monitors embedded controller interactions with external CPC components to ensure system-level specifications of stability and security are maintained.

More approaches like TAIGA are needed as it is increasingly clear that security for embedded systems is paramount in CPCs. Embedded security should not be a hand-me-down from approaches developed for networks or general-purpose computing platforms. It should also not be an afterthought when developing CPCs. Rather, it is up to designers and upper-level decision makers to adopt best security principles, practices, and architectures specific to their CPC application during system development.

An area of ongoing research is TAIGA implementation and efficiencies in a variety of CPC environments. Complex CPCs often have many control loops which must function in a coordinated fashion. Architectural modifications may be required to scale and distribute TAIGA among multiple embedded systems in such an environment. Alternatives to implementing TAIGA alongside during the controller development process should also be investigated. For example, TAIGA integration with existing, third-party CPC controllers should be addressed. Alternatives to programmable logic-based TAIGA implementations might be more appropriate in some cases. The possibility of applying TAIGA to other domains outside of run-time CPC protections should also be investigated. For instance, TAIGA also has a potential system development and analysis use by providing real-time interface observability and controllability.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number CNS-1222656. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We are grateful for donations from Xilinx, Inc. and support from the Georgia Tech Research Institute.

References

[1] Miron Abramovici and Paul Bradley. Integrated circuit security: new threats and solutions. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 55:1–55:3, New York, NY, USA, 2009. ACM.

[2] Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The MILS Architecture for High-Assurance Embedded Systems. *International Journal of Embedded Systems*, 2:239–247, 2006.

[3] ARM Ltd. TrustZone technology, 2014. `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[4] Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 34–39, June 2011.

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, August 2011.

[6] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in: Fluorine-20130501*. CEA LIST, Software Reliability Laboratory, Saclay, F-91191, 2013. `http://frama-c.com`.

[7] C. Dai, S.H. Yang, and Liansheng Tan. An approach for controller fault detection. In *Fifth World Conference on Intelligent Control and Automation (WCICA)*, volume 2, pages 1637–1641, Jun 2004.

[8] Luke Dubord. Waiting on the speed of light: Engineering autonomy at Mars. In *DESIGN West Conference*, San Jose, CA, April 2013.

[9] N. Falliere, L. O'Murchu, and E. Chien. W32.stuxnet dossier, 2011.

[10] Zane R. Franklin, Cameron D. Patterson, Lee W. Lerner, and Ron J. Prado. Autonomic hardware for trust enhancement of critical embedded processes. In *7th International Symposium on Resilient Control Systems (ISRCS)*, Denver, CO, August 2014.

[11] Ted Huffmire, Timothy Levin, Thuy Nguyen, Cynthia Irvine, Brett Brotherton, Gang Wang, Timothy Sherwood, and Ryan Kastner. Security primitives for reconfigurable hardware-based systems. *ACM Trans. Reconfigurable Technol. Syst.*, 3:10:1–10:35, May 2010.

[12] Intel. Intel Trusted Execution Technology architectural overview, 2007. http://www.intel.com/technology/security/downloads/arch-overview.pdf.

[13] Intel. Intel Software Guard Extensions programming reference, 2014. https://software.intel.com/sites/default/files/managed/48/88/ 329298-002.pdf.

[14] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46, Oct. 2010.

[15] Lee W. Lerner, Mohammed M. Farag, and Cameron D. Patterson. Run-time prediction and preemption of configuration attacks on embedded process controllers. In *International Conference on Security of Internet of Things (SecurIT 2012)*, Kerala, India, Aug 2012.

[16] Lee W. Lerner, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson. Application-level autonomic hardware to predict and preempt software attacks on industrial control systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, June 2014.

[17] Claude Marché and Yannick Moy. *The Jessie plugin for Deductive Verification in Frama-C*. INRIA Saclay - Télé-de-France and LRI, CNRS UMR 8623, 2013. `http://proval.lri.fr/`.

[18] Mark McLean and Jason Moore. Securing FPGAs for red/black systems: FPGA-based single chip cryptographic solution. *Military Embedded Systems Mag.*, 2007.

[19] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. In *Black Hat USA*, July 2014.

[20] National Institute of Standards and Technology (NIST). NVD Common Vulnerability Scoring System Support v2, June 2007. http://nvd.nist.gov/cvss.cfm?calculator.

[21] Dan Rosenberg. QSEE TrustZone kernel integer overflow vulnerability. In *Black Hat USA*, July 2014.

[22] William Safire. The Farewell Dossier. *New York Times*, Feb 2, 2004.

[23] Moses Schwartz, John Mulder, Adrian R. Chavez, and Benjamin A. Allan. Emerging techniques for field device security. *Security Privacy, IEEE*, 12(6):24–31, Nov 2014.

[24] Lui Sha. Using simplicity to control complexity. *Software, IEEE*, 18(4):20–28, Jul–Aug 2001.

[25] Agam Shah. Heartbleed exposes weaknesses in hardware design. *PCWorld*, 2014.

[26] Microsoft TechNet. Microsoft Next-Generation Secure Computing Base - Technical FAQ, July 2003. http://technet.microsoft.com/en-us/library/cc723472.aspx.

[27] Mohammad Tehranipoor, Hassan Salmani, Xuehui Zhang, Michel Wang, Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. Trustworthy hardware: Trojan detection and design-for-trust challenges. *Computer*, 44(7):66–74, July 2011.

[28] Andr Teixeira. *Toward Cyber-Secure and Resilient Networked Control Systems.* PhD thesis, KTH Royal Institute of Technology, 2014.

[29] Trusted Computing Group. TPM main specification, 2015.

[30] Mark Zeller. Myth or reality – does the Aurora vulnerability pose a risk to my generator? In *Protective Relay Engineers, 2011 64th Annual Conference for*, pages 130–136. IEEE, 2011.

Chapter 3

Isolating Trust in an Industrial Control System-on-Chip Architecture

3.1 Abstract

A distributed industrial control system (ICS) also distributes trust across many software and hardware components. There is a need for some malware countermeasures to be independent of application, supervisory or driver software, which can introduce vulnerabilities. We describe the Trustworthy Autonomic Interface Guardian Architecture (TAIGA) that provides an on-chip, digital, security version of classic mechanical interlocks. In order to enhance trust in critical embedded processes, TAIGA redistributes responsibilities and authorities between a Programmable Logic Controller (PLC) processor and a hardware-implemented interface controller, simplifying PLC software without significantly degrading performance while separating trusted components from updatable software. The interface controller is synthesized from C code, formally analyzed, and permits runtime checked, authenticated updates to certain system parameters but not code. TAIGA's main focus is ensuring process stability even if this requires overriding commands from the processor or supervisory nodes. The TAIGA architecture is mapped to a commercial, configurable system-on-chip platform.

3.2 Introduction

Many technologies developed initially for personal and IT platforms eventually appear in high-performance embedded systems. For example, architectural enhancements such as multi-core processors, heterogeneous accelerators, multi-level caches, and TCP/IP network interfaces are now common in state-of-the-art embedded platforms. Malware is also migrating from computer systems used mostly for exchanging information and processing data to PLCs controlling physical processes. Unfortunately IT and personal device perimeter defenses are not a sufficient countermeasure since intrusion protection is initially reactive rather than preventative, and PLC platforms typically do not incorporate sophisticated network defenses. Verification is not significantly easier due to the overall embedded system complexity and periodic software updates from many sources. Yet the main distinguishing feature of control systems — a singular focus on regulating the state of physical processes — is not normally exploited when trying to enhance security and trust. In both environments supervisory code may be responsible for managing system resources and processes. However this code does not enforce process-specific rules, or any rules at all in the case of a simple real-time kernel such as FreeRTOS.

As illustrated in Fig. 3.1, the ultimate objective of ICS malware is PLC behavior modification from: (1) process-harming Master Terminal Unit (MTU) or human operator commands; (2) false data injection; or (3) updates to PLC code. Replicating SCADA network gateways [5] and MTUs [14] are examples of countermeasures for MTU-directed attacks. Compromised communication between system components may prevent a PLC from maintaining process stability, although physical security may mitigate this threat. Stuxnet, a virus designed to tamper with PLCs controlling centrifuges, demonstrated a powerful attack vector via changes to controller code [4]. Changes to both PLC software and firmware (supervisory code and drivers) need to be considered [24]. We address all three classes of attacks, and our approach is distinguished by adding malware resilience to the PLC rather than seeking to prevent attacks from ever reaching the PLC.

With the advent of configurable system-on-chip (SoC) platforms combining microcontrollers with programmable logic, we propose trust enhancement of critical embedded processes at a level below even the kernel, namely a self-contained I/O processor. The Trustworthy Autonomic Interface Guardian Architecture (TAIGA) is the sole means by which a PLC's general-purpose processor (GPP) communicates through any external interface to sensors,

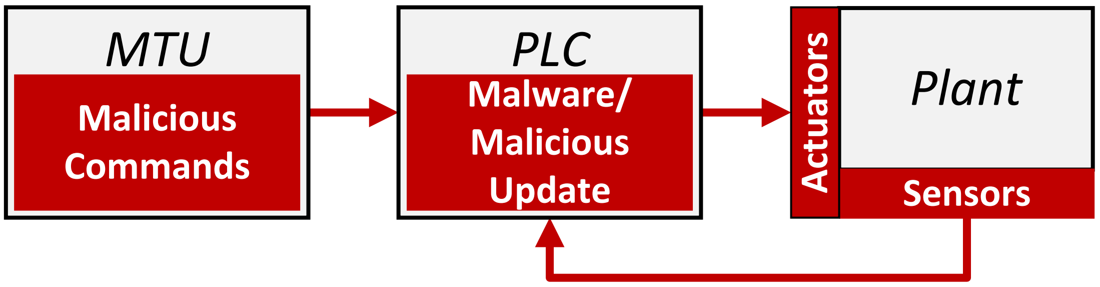


Figure 3.1: Threats to programmable process controllers

actuators, and networks. All external traffic is monitored by the TAIGA, which can override GPP actions if a physical process is deviating from specifications. To mitigate a Stuxnet-like attack, the TAIGA can take corrective action (such as reverting to a backup controller) and generate alarms without GPP assistance should a controlled physical process risk instability due to malware-infected GPP code, malicious requests from MTUs, or sensor malfunctions. Hence the TAIGA acts as a physical process guardian by enforcing formally validated safety and liveness properties that are independent of GPP software actions or inactions. Commercially available configurable SoC platforms enable GPP software executed on ARM cores and TAIGA implementation in programmable logic. The TAIGA's mapping to hardware allows the integration of security and I/O processing without incurring unacceptable performance, power, or size trade-offs. GPP software is simplified since serial bus and network interface device drivers are no longer needed.

In addition to offloading I/O, security, and trust responsibilities from GPP software, TAIGA modifications are more controlled than GPP software changes. While remote or GPP-directed updates to the configurable logic are disallowed, an encrypted, network-facing maintenance port allows restricted, authenticated updates to a subset of system parameters. The GPP is not involved in these transfers since the GPP connects to the network interface through the TAIGA. After decryption and authentication using TAIGA-internal keys and cryptographic modules, TAIGA checks parameter update values using static code that goes through the same formal validation steps as process monitoring code. An example of a permitted update is a range-limited model coefficient adjustment to accommodate physical resource (e.g. motor) aging. On the other hand, changes to code such as the backup controller are disallowed and would require replacement of the configurable logic bitstream defining the TAIGA. This is in sharp contrast to GPP updates that may change any data, software or firmware.

TAIGA's abstract organization, concrete implementation, and update safeguards are presented in Section 3.3. Section 3.4 illustrates TAIGA protections for a simple motor controller running on a Xilinx Zynq-7000 SoC. Finally, Section 3.5 summarizes completed and ongoing work.

3.3 TAIGA Overview

3.3.1 Interfaces and Internal Structure

As shown in Fig. 3.2, TAIGA is a module serving as the sole path between a PLC microcontroller's internal bus and peripheral controllers. Hence it has two ports: a bus slave interface to GPPs, and a bus master interface to peripherals. Neither GPPs nor peripheral controllers require modification. Architecturally the TAIGA has similarities to I/O channel processors still used on mainframe computers, and the overall system organization remains consistent with modern SoC design. All communication between GPPs and external serial buses (such as I2C, SPI, or UARTs) and the network interface (such as Ethernet) are routed through the TAIGA, which is also responsible for initializing peripheral controllers. While a single logical port is presented to GPPs, the TAIGA can have multiple memory-mapped physical ports on the GPP's bus and individual connections to one or more peripheral controllers. As a result the TAIGA does not necessarily degrade I/O concurrency compared to a conventional organization in which the GPPs communicate with all peripherals through a single shared bus, although I/O latency may increase.

TAIGA's internal blocks are listed in Table 3.1. Methods of implementation depend on the target technology, with one mapping described in Section 3.3.2.

3.3.2 Implementation on a Commercial Configurable SoC

Xilinx's Zynq-7000 All Programmable SoC integrates both a Processing System (PS) and Programmable Logic (PL) on a single 28nm chip [10]. The PS uses an AXI bus to connect an ARM Cortex-A9 dual-core GPP, two levels of cache, on-chip memory, internal timers, and a large collection of controllers for peripherals such as external memories, serial buses, and Ethernet. The PL uses programmable segmented routing to connect hardware resources

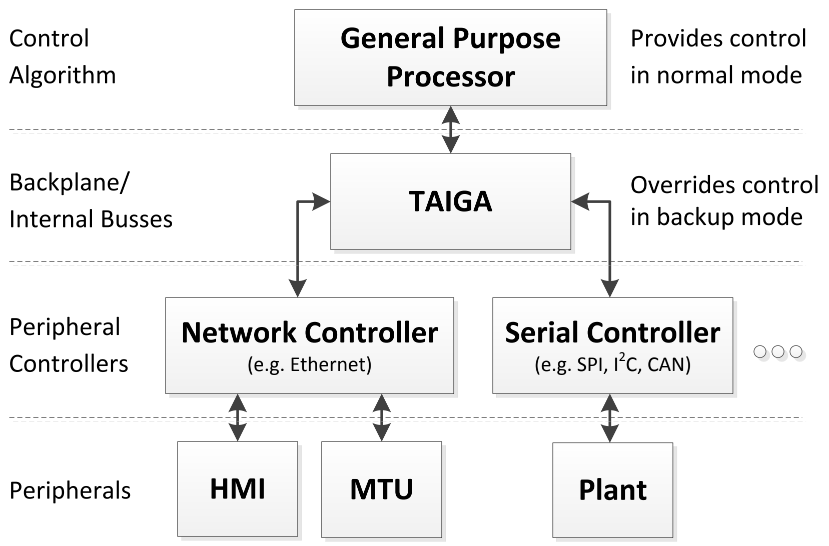

Control Algorithm
General Purpose Processor
Provides control in normal mode
Backplane/ Internal Busses
TAIGA
Overrides control in backup mode
Peripheral Controllers
Network Controller (e.g. Ethernet)
Serial Controller (e.g. SPI, I2C, CAN)
Peripherals
HMI
MTU
Plant

Figure 3.2: TAIGA insertion in a generic PLC microcontroller

such as configurable logic blocks, block memories, arithmetic units, clock managers, and I/O blocks. The PS can operate independently of the PL, but these two subsystems can also communicate in several ways.

Fig. 3.3 shows a PL insertion of the TAIGA. There is only one overall AXI bus allowing the GPPs to communicate directly with peripheral controllers without going through the TAIGA. This mismatch with Fig. 3.2 does not prevent a TAIGA implementation, and it is straightforward to generate addressing exceptions if the GPPs issue addresses in the range reserved for peripheral controllers. Looking at Table 3.1, the PL already has pairs of 32-bit AXI master and slave interfaces. In order to implement the Ethernet controller's IP stack, the master interface incorporates a MicroBlaze soft processor. All other TAIGA components, except the junction box, are captured in ANSI C code and implemented in the PL using Xilinx's high-level synthesis (HLS) tool. HLS support of floating point simplifies the specification guards, plant model, backup controller, and monitor. The junction box is implemented in the Verilog hardware description language since it mostly implements 32-bit connections between the other components.

Security ramifications of using a PS core and memory to implement prediction need to be considered. Advantages include performance and complete production controller emulation including firmware such as FreeRTOS. Prediction's plant model and specification guards reside in the TAIGA and cannot compromised by malware. At worst the prediction controller

Table 3.1: TAIGA internal components

Name	**Use**
Bus slave unit(s)	Connects to GPPs
Bus master unit(s)	Connects to peripheral controllers
Specification guards	Decide if plant sensor readings have valid ranges
Plant model	State prediction and sensor integrity checks
Backup controller	Preserves plant stability
Monitor	Checks the specification guards, current plant state, and predicted plant model state to decide whether the backup controller should override the GPP's production controller
Cryptographic unit	Encrypt/decrypt and authenticate blocks using internal keys
Parameter update	Uses the crypto unit to read process parameter update requests and write them to flash memory
Junction box	Connects the above components

can cease to function correctly or attempt to interfere with the production controller, thereby invoking the backup controller.

3.3.3 Parameter Update Process

TAIGA protections could be nullified if unrestricted updates to the PL configuration are allowed. We assume ICS physical security or else direct sabotage of the plant or actuators would be possible. Zynq-7000 boards can prevent PS-initiated reconfiguration of the PL by requiring jumper changes to load an AES-encrypted bitstream into flash memory. The bitstream encryption key must match the decryption key stored in non-volatile memory on the Zynq device.

While remote bitstream updates to TAIGA control logic are disallowed, there is still a need to support remote maintenance updates to system-level constants and coefficients used by the specification guards, plant model, and backup controller. These constants form a system parameter set whose current, minimum and maximum values are stored in TAIGA PL resources. When a parameter update block is received from an MTU, the cryptography

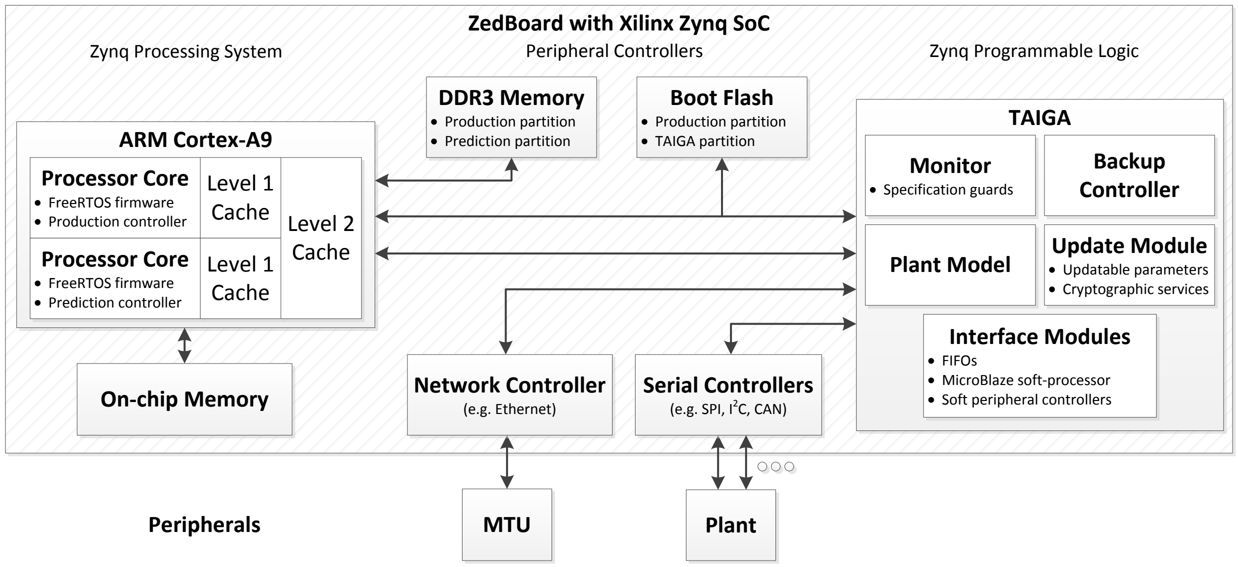


Figure 3.3: TAIGA implementation on the Zynq-7000 SoC

unit decrypts and authenticates the block. As illustrated in Section 3.4.1, the parameter update module uses formally analyzed code to ensure new values are acceptable. Hence the TAIGA does not rely only upon cryptographic methods and has the final authority to accept parameter updates. Validated blocks are copied to the system parameter set and the current value revisions are written as an encrypted and authenticated block on external flash.

3.4 Motor Control Example

A simple motor control application adapted from [3] is used to test and demonstrate TAIGA effectiveness. In this implementation the production controller is a proportional-integral-derivative (PID) controller and the backup controller is a proportional-only controller with a fixed reference input to ensure stable motor operation. The motor itself is emulated in software functions, and its speed is sampled once per millisecond. Specification guards monitor the average speed of the motor over a given period. The MTU can securely modify minimum and maximum limits for this average only with a formally verified update function described in Section 3.4.1.

3.4.1 System Parameter Update Analysis

The HLS design flow enables high-level formal analysis tools to aid the validation and verification process. An abstracted verification model is not required and the C source is analyzed directly. Additionally, security property reasoning reduces the verification space from the entire system to only TAIGA additions. We formally validate TAIGA code functionality using Frama-C, an open source, modular static analysis framework for the C language [9]. In our previous work Frama-C performed deductive reasoning on TAIGA code to verify proofs of specification guard and backup condition functionality [7]. In this work Frama-C's Value Analysis confirms the TAIGA only accepts updated system parameters that are within internally specified minimum and maximum values.

Fig. 3.4 contains a code excerpt from the TAIGA's system parameter update module. The first 5 lines define parameter identifiers and structures to hold minimum, maximum, and current values for each data type. These structures are instantiated for each updatable value, as shown in the next 3 lines. The update functions in the remaining code check new requested values against the non-updatable `min` and `max` values stored in the structures before allowing current values to be updated.

Frama-C's Value Analysis plug-in computes sets of all possibles values that program variables can take. The code can be analyzed as written without the need for additional proof annotations. Fig. 3.5 shows the value analysis for the update functions from Fig. 3.4. The analysis confirms the current values of the `y_min`, `y_max`, and `probation` variables are kept within the allowed ranges. Note that the range notation for some `const` fields arises from floating point imprecision.

3.4.2 System Operational Flow

Fig. 3.6 is a schematic showing the integration of TAIGA with the motor controller. This system repeatedly performs the following sequence of steps:

1. The motor produces new sensor data once per millisecond.
2. The production and backup controllers generate new control data based on the motor's output.

```
enum Id {Y_MIN, Y_MAX, PROBATION};
struct SP_float { const float min, max;
                  float curr; };
struct SP_int { const int min, max;
                int curr; };
struct SP_float y_min = {-3.2f, -0.1f, -2.0f};
struct SP_float y_max = {0.0f, 3.2f, 2.0f};
struct SP_int probation = {100, 1000, 200};

void update_SP_float(enum Id id, float val) {
 if (id==Y_MIN) {
  if ((val>=y_min.min) && (val<=y_min.max))
   y_min.curr = val;
 } else if (id==Y_MAX) {
  if ((val>=y_max.min) && (val<=y_max.max))
   y_max.curr = val;
 }
}

void update_SP_int(enum Id id, int val) {
 if (id==PROBATION) {
  if ((val>=probation.min) &&
      (val<=probation.max))
   probation.curr = val;
 }
}
```

Figure 3.4: TAIGA system parameter update code

3. Specification guards examine the past average speed of the motor and the predicted average speed of the model from the previous control cycle. If these values are within range, the production controller's output is passed to the physical motor. If not, the backup controller's output is passed to the physical motor.

4. If the backup condition has been triggered, selector `B` passes the backup control data to the model of the motor for confirmation of sensor data.

5. The past average speed of the motor is updated with the new sensor data.

6. The current state of the model is saved.

7. The prediction controller and motor model run for N cycles as a virtual control loop

```
Values for function update_SP_float:
    y_min.min  ∈ [−3.200000 .. −3.199999]
         .max  ∈ [−0.100000 .. −0.099999]
         .curr ∈ [−3.200000 .. −0.099999]
    y_max.min  ∈ {0.0}
         .max  ∈ [3.199999 .. 3.200000]
         .curr ∈ [0.0 .. 3.200000]
Values for function update_SP_int:
    probation.min  ∈ {100}
             .max  ∈ {1000}
             .curr ∈ [100 .. 1000]
```

Figure 3.5: Frama-C Value Analysis results

in order to approximate the future behavior of the motor.

8. The predicted average speed of the motor is calculated.

9. The saved state of the model is restored, and the system waits for new sensor data.

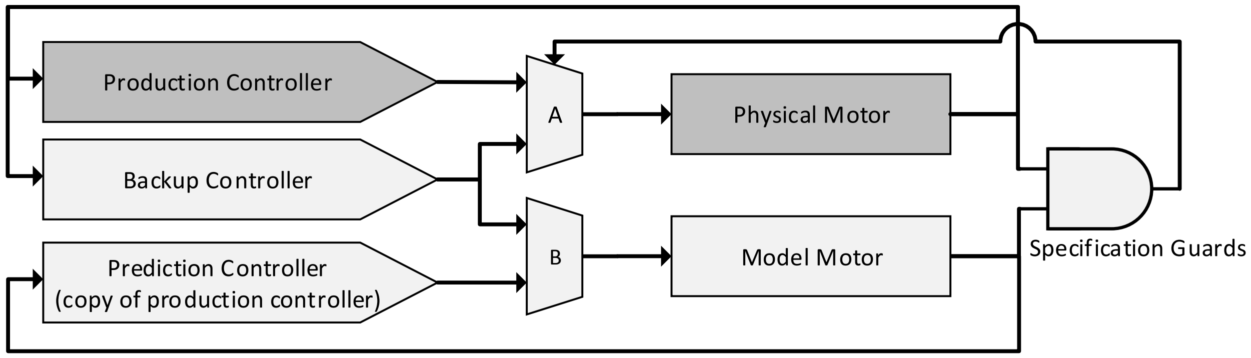


Figure 3.6: Motor control system with TAIGA safeguards

Production control code runs on the Cortex-A9 GPP. The prediction controller is an identical copy of the production controller. For this work a single Cortex-A9 core is multiplexed to operate both the production and prediction controllers. If separation is desired the prediction controller may be moved to the second GPP core. All other components are implemented in the PL configurable hardware including a software emulation of the physical motor on a MicroBlaze soft processor.

3.4.3 Controller Malware Response

A simulated sequence of events is used to demonstrate the effectiveness of TAIGA in a motor control application. In this example, a prediction window of 200 ms is used. Specification guards monitor the average speed of the motor over a 400 ms period (200 ms past, 200 ms predicted). Each time the specification guards detect a fault, a probation counter is set. During this probation period, the production controller's output is ignored even if it returns to an acceptable value. If the production controller output remains within a valid range throughout the probation period, it becomes eligible for reinstatement. For this example, a probation period of 150 ms is used.

Fig. 3.7 displays the motor's response to certain events. The sequence begins with a unit step response and a maximum allowable average speed (`y_max`) of 2.0 (in normalized units). At $t = 250$ ms, the reference input to the production controller changes from 1.0 to 2.5. The predicted model output indicates the average speed of the motor will exceed `y_max` (2.0) within 200 ms. Therefore the backup controller assumes control and moves the motor to a stable output of 1.3.

This increase in motor speed is deemed necessary and `y_max` is increased to 2.75 at $t = 300$ ms to accommodate this speedup. Production control outputs are now within acceptable range but the probation period does not expire until 150 cycles later at $t = 450$ ms. Once the probation ends, the production controller is reinstated and increases the motor speed to 2.5. Motor speed surpasses `y_max` (2.75) due to overshoot at $t = 518$ ms, but the backup controller is not invoked because the average speed does not exceed `y_max`. Production controller operation continues until $t = 700$ ms, at which time its reference input is increased to 5.0. The motor cannot safely operate at this speed, and is instead returned to a known stable speed by the backup controller.

TAIGA contains internal timers which monitor the response times of the production and prediction controllers. In this example if either fails to respond within the allowed time, the backup condition is triggered and a reset is required to return the system to normal operation. This mechanism allows the TAIGA to maintain stable motor operation in the event of component failure or communication interruption. The response to the loss of the production or prediction controller is application-specific.

Prediction window size may be limited by the target platform's performance and by the model's accuracy. The size and ratios of the period over which the average speed of the motor is calculated are application-dependent. A larger period may be desired, or more weight given to past values. The given probation period was chosen to be smaller than the prediction window in order to more quickly return to full operation. This is especially useful in systems frequently targeted by denial-of-service attacks. For critical systems, however, this period should be extended or a more robust system for safely reinstating the production controller may be implemented. Alternatively it may be desirable to have no means of reinstatement and instead require a reset before returning to full operation.

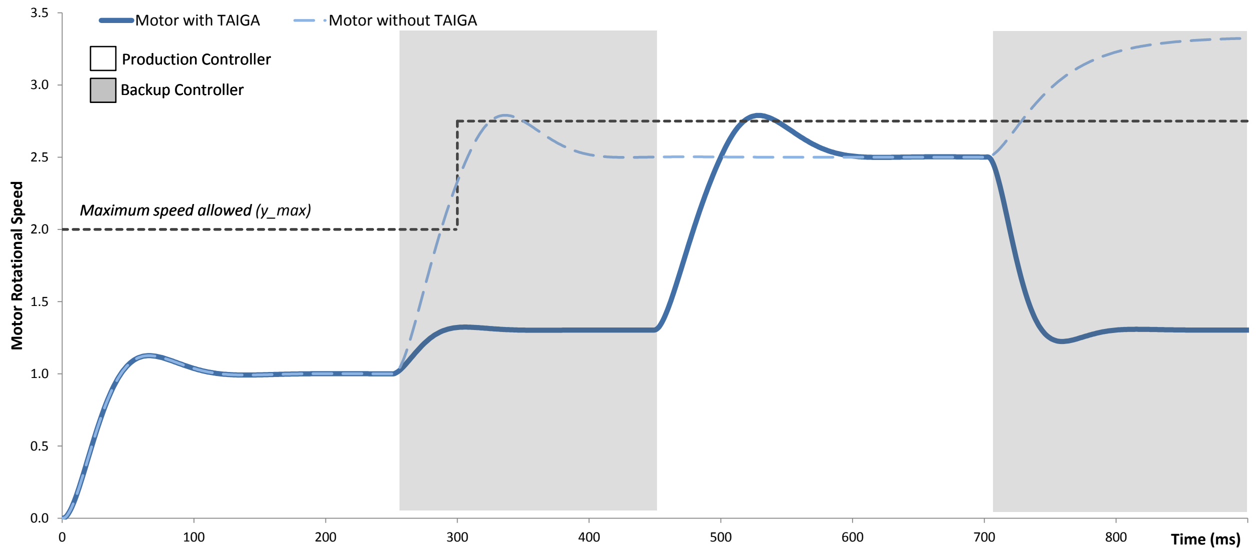


Figure 3.7: TAIGA response to malware in a motor control application

3.4.4 Time and Resource Analysis

Table 3.2 provides the latency of critical timing paths. The Cortex-A9 GPP operates at 667 MHz while its programmable logic is clocked at 130 MHz. As listed, the runtime of one iteration of the prediction control loop is 1660 ns. However, I/O and communication protocols consume roughly 72% of this latency, as the combined runtime of the prediction controller and model is only 472 ns. Based on this runtime, the system can safely predict over 500 control cycles into the future.

On average, TAIGA requires roughly 1 μs to complete its operations. This additional latency should normally be acceptable for a system with a 1 ms sensor sampling rate. In addition, a 1 μs latency is modest compared to implementing TAIGA functions with software, network communication, or a separate chip. The time required by the system to produce a new control command after receiving new sensor data is approximately 2 μs. Considering the runtimes of the production controller and TAIGA, 66% of this latency is due to I/O and communications.

Table 3.2: Timing analysis

Timing path	**Latency (ns)**
Production controller runtime	280
Model runtime	192
TAIGA runtime	1040
One iteration of prediction control loop	1660
Sensor sample $\rightarrow$ New control command	2000

TAIGA logic resource consumption is shown in Table 3.3. Two BRAMs are required for tracking past values of sensor data, and multiple floating point units utilize 10 DSP slices. The junction box's footprint is negligible as its primary function is simply monitoring, passing and redirecting signals. The MicroBlaze soft processor, which functions as the TAIGA's bus master component, is responsible for the majority of the resources used. For example, 64KB of processor local memory requires 15 BRAMs, and a floating point unit uses 4 DSP slices. Total resource consumption does not exceed 20% for any particular resource type. This utilization is relatively modest as the Zynq-7000's PL is a fraction of that available in high density FPGAs. An abundance of unused resources would permit further TAIGA latency reduction using HLS time/area tradeoff directives. Relative to a microprocessor with a single monolithic memory, HLS can improve performance through the parallel execution of code accessing data structures mapped to independent multi-ported memories.

Table 3.3: Programmable logic resource usage

	FF	LUT	DSP	BRAM
TAIGA HLS components	4321	6229	10	2
Junction box	139	205	0	0
Total with MicroBlaze	10579	10260	14	17
Available	106400	53200	140	220
Utilization	10%	19%	10%	8%

3.5 Conclusions and Current Work

Our overarching research goal is platform-based architectural support for, and rigorous validation of, Trust Enhancement of Critical Embedded Processes (TECEP) [7, 16]. We are guided by a design philosophy that the most trusted layers of a system should validate requests from less trusted layers, and otherwise take corrective actions. While this may not be easily accomplished in systems interacting with people, process control has the advantage of precise specifications and accurate models that can be used to check the immediate or eventual consequences of a controller's malicious or inadvertent actions. Our prior work sought to make these enhancements transparent to any GPP code, including the OS. While successful, resource duplication and the lack of a controlled update mechanism were deficiencies addressed by TAIGA.

Many embedded systems tailor the platform architecture to the application, with the Zynq-7000 offering a great advance in this respect. TAIGA uses this flexibility for improving security and trust rather than only performance and power. Many security and trust extensions add external hardware components which would complicate ICS reliability analysis, or internal software layers which likely exceed the capabilities of formal analysis tools. In contrast TAIGA improves resilience of the existing PLC node and simplifies its GPP software by removing device drivers. While network security measures are necessary in an ICS to protect MTUs, PLCs are the bridge between the cyber and physical worlds and should have self-contained safeguards such as TAIGA.

Ongoing work includes observing that production and prediction controllers are nearly interchangeable since both get sensor data and write actuator commands through opaque FIFOs. In addition to prediction, this symmetry could vet software updates first with a plant model, and provide a hot-standby controller. Update control mechanisms are also being developed for other domains such as the Internet of Things. Finally, we are transitioning from simulated results to physical experimentation with a Zynq-based motor controller [1] running on GTRI's ICS Security Test Bed.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number CNS-1222656. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We are grateful for ZedBoard and tool donations from Xilinx, Inc.

References

[1] Avnet. Xilinx Zynq-7000 All Programmable SoC/Analog Devices Intelligent Drives Kit, 2013. Avnet product brief.

[2] Daniele Bagni and Duncan Mackay. Floating-point PID Controller Design with Vivado HLS and System Generator for DSP, January 2013. XAPP1163 (v1.0).

[3] A.N. Bessani, P. Sousa, M. Correia, N.F. Neves, and P. Verissimo. The Crutial way of critical infrastructure protection. *Security Privacy, IEEE*, 6(6):44–51, Nov 2008.

[4] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in: Fluorine-20130501.* CEA LIST, Software Reliability Laboratory, Saclay, F-91191, 2013. `http://frama-c.com`.

[5] N. Falliere, L. O'Murchu, and E. Chien. W32.stuxnet dossier, 2011.

[6] J. Kirsch, S. Goose, Y. Amir, Dong Wei, and P. Skare. Survivable SCADA via intrusion-tolerant replication. *Smart Grid, IEEE Transactions on*, 5(1):60–70, Jan 2014.

[7] Lee W. Lerner, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson. Application-level autonomic hardware to predict and preempt software attacks on industrial control systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, June 2014.

[8] Lee W. Lerner, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson. Using high-level synthesis and formal analysis to predict and preempt attacks on industrial control systems. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 209–212, Monterey, CA, Feb 2014.

[9] Carl Schuett, Jonathan Butts, and Stephen Dunlap. An evaluation of modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 7(1):61 – 68, 2014.

[10] Xilinx. Zynq-7000 All Programmable SoC, February 2014. UG585 (v1.7).

Chapter 4

Application-level Autonomic Hardware to Predict and Preempt Software Attacks on Industrial Control Systems

4.1 Abstract

We mitigate malicious software threats to industrial control systems, not by bolstering perimeter security, but rather by using application-specific configurable hardware to monitor and possibly override software operations in real time at the lowest (I/O pin) level of a system-on-chip platform containing a microcontroller augmented with configurable logic. The process specifications, stability-preserving backup controller, and switchover logic are specified and formally verified as C code commonly used in control systems, but synthesized into hardware to resist software reconfiguration attacks. In addition, a copy of the production controller task is optionally implemented in an on-chip, isolated soft processor, connected to a model of the physical process, and accelerated to preview what the controller will attempt to do in the near future. This prediction provides greater assurance that the backup controller can be invoked before the physical process becomes unstable. Adding trusted, application-tailored, software-invisible, autonomic hardware is well-supported in a commercial system-on-chip platform.

4.2 Introduction

The dark side to an increasingly automated physical world is the potential for large scale disruption and destruction caused by the malicious reprogramming of process control systems. Militaries and extremist groups are aware of these vulnerabilities and opportunities, and the geopolitical effects may be as significant as the development of nuclear weapons. There is a summative effect from the small (individual process control systems) to the large (an electric grid). Power generation systems, in particular, have demonstrated vulnerabilities, and cannot be quickly repaired or replaced [5]. Descriptions of industrial control system (ICS) zero-day exploits can be purchased from hackers through on-line brokerages [19]. It is not clear how to discourage these attacks since anyone with Internet access can develop and deploy cyberweapons, a country's military offers little protection of manufacturing and infrastructure from cyberattacks, an attack's ultimate source may be difficult to determine, and the threat of retaliation may not be an adequate deterrent. While ICS attacks are considered a recent threat fast becoming "an element of almost any crisis" [23], there is speculation that a Siberian gas pipeline was sabotaged in 1982 by CIA-directed implantation of malware causing excessive gas pressure [22]. The explosion had roughly one-sixth the power of the first atomic bomb.

There are similarities but also important differences between protecting information and safeguarding physical infrastructure [26]. ICSes structured as Supervisory Control and Data Acquisition (SCADA) systems have hierarchical, networked computer infrastructure and human-computer interaction, as well as edge nodes consisting of ruggedized microcontrollers called Remote Terminal Units (RTUs) or Programmable Logic Controllers (PLCs) that bridge the cyber and physical worlds [7]. (Henceforth references to PLCs also includes RTUs.) The non-edge PC nodes such as Master Terminal Units (MTUs) monitor, coordinate, optimize, and update the PLCs. This mixture of conventional and embedded computing platforms enables attack vectors common to IT environments while also exposing microcontrollers running simple kernels designed for real-time performance rather than intrusion protection. Many ICS attacks focus on network protocol and supervisory platform exploits to modify commands given to PLCs. Modifying PLC firmware or control code achieves the same ends [24].

Protecting manufacturing, infrastructure, and future cyber-physical systems requires unqualified resistance to cyberattacks created by adversaries with nation-scale resources. While robust perimeter security and supervisory node intrusion resilience are necessary and should be the primary defense, we consider the worst-case scenario where SCADA networks have been penetrated, MTUs have been compromised, and even PLCs have been reprogrammed [15]. This may be accomplished through one or more avenues: insiders, manufacturer's maintenance ports, third party code, zero-day exploits, and PC updates. Air gaps and effective physical security measures are not a sufficient defense, as evidenced by Stuxnet [4]. TECEP (Trust Enhancement of Critical Embedded Processes) is our method of ensuring process stability in a way that that does not rely on trust in any software layer on any ICS node. Isolation is achieved using high-level synthesis (HLS) targeting configurable hardware in a commercial system-on-chip IC used to implement the PLC's microcontroller [16]. Predicting future deviation from normal operation is enabled by reuse of specifications for normal system operation and accurate models for the physical process. Normally these models are used only during development, but TECEP retains them in the fielded platform.

While using redundancy to cope with sensor faults is a reasonably well-solved problem, malicious software attacks are a new and distinct area of concern [8]. Middle ground is needed between viewing ICS cybersecurity as solely a network- or supervisory-level concern, and overburdening control system engineers with additional roles. Our approach tries to accomplish this through integration with a conventional model-based design flow, synthesizing and formally verifying hardware-implemented functions to monitor the current and future state of physical process, and switching to a stability-preserving backup controller if necessary. HLS avoids the added complexity of using hardware description languages, and modest performance requirements permit hardware generation without iterative timing optimization.

Section 4.3 describes the conventional ICS development process, and summarizes existing ICS-specific strategies used to enhance security. TECEP's system-on-chip platform organization, and synthesis/analysis extensions to the model-based design flow are presented in Section 4.4. Section 4.5 illustrates and assesses TECEP with the synthesis and verification of monitoring, predictive and preemptive enhancements to a simple motor controller. Finally, Section 4.6 summarizes completed and ongoing work.

4.3 Current Approaches to ICS Development and Security

PLCs periodically read sensor data $y(t)$ from a physical process (referred to as the *plant*), compute the error $e(t)$ between $y(t)$ and the desired plant state $w(t)$, and adjust plant inputs $u(t)$ (such as mechanical actuators) in order to minimize $|e(t)|$. A simple example is adjusting the fuel supply to a furnace when the furnace temperature changes. Commonly used PID controllers compute $e(t)$ by summing proportional (present error), integral (past error), and derivative (future error) terms [7].

Control system engineering has enjoyed an enviable marriage of theory and practice, and is one of the most widely adopted examples of model-based design which has the following steps:

1. Create a mathematical model of the plant either from physical laws or by acquiring and analyzing real data.
2. Use the plant model to develop an effective control algorithm.
3. Simulate the response of the system to inputs such as a unit step change on $w(t)$.
4. Synthesize controller code for a particular embedded platform.

MATLAB/Simulink supports this methodology, and generates code with the Embedded Coder toolbox [18].

4.3.1 ICS-independent Security Techniques

Hardware and software architectures trickle down from personal computing systems to embedded platforms in order to match capabilities. Unfortunately threats also migrate, and the defenses used to protect information are also needed to protect control processes. We do not consider side-channel or fault-injection attacks often associated with embedded platforms since if one already has physical access to a controller then there would be a more direct means of degrading or destroying the physical processes. Rather, the main attack vector to be mitigated is unsanctioned use of a network for the purposes of sending new commands or software to a PLC while perhaps simultaneously reporting normal operating status to human operators. This was precisely the *modus operandi* for Stuxnet.

Most defenses define one or more zones of trust, with increased trust granted to the software and hardware inside a given zone. Inner zones typically provide services to outer zones as in the case of an operating system kernel. Hardware mechanisms may help to prevent the unauthorized use of resources inside a trusted zone. The first security architecture described in Table 4.1 requires privileged operations, such as hardware and process management, to be performed by a single kernel such as Linux. As the number of services increases, however, the complexity of the service-providing code makes it likely that bugs exist that can be probed in an automated way and possibly used to the adversary's advantage. Potential weaknesses in this scheme include: all applications run in the same zone, the kernel must be all things to all applications, and reliance on defenses such as firewalls receiving regular updates. According to Eugene Spafford, firewalls were originally introduced as a stopgap measure until host security was improved [13].

System architecture extensions may be provided to reduce the attack surface for a subset of applications. For example, TrustZone is ARM's extension allowing certain applications to run in a secure zone (SZ) running under a distinct kernel in a separate memory space and able to access particular hardware resources unavailable to applications that run in the normal zone (NZ) [1]. Both zones share the same physical processor(s), which execute monitor mode code when switching between zones. Although the SZ's kernel may be simpler than the NZ's kernel, considerable complexity and possible exploits likely remain. For example, a vulnerability in the SZ kernel was used to jailbreak an Android device [21].

As seen above, the standard approach for enhancing the security of platforms required to run a variety of applications has been to impose or extend a set of hardware-enforced, restrictive processor modes. However, another approach may be used if the platform is dedicated to domain-specific applications such as process control. The current generation of Field-Programmable Gate Arrays (FPGAs) permit functions to be implemented as software running on hard or soft processors, or directly in custom hardware. Normally this capability is used to improve system performance and/or power attributes, perhaps by increasing the time and power efficiency of cryptographic functions. As previewed in the third row of Table 4.1 and described further in Section 4.4, TECEP instead exploits this flexibility to invert software and hardware authorities by implementing critical ICS functions in formally verified PLC hardware blocks that cannot be controlled or modified by any local or remote

Table 4.1: Separating a system-on-chip (SoC) platform into a normal zone (NZ) and a secure zone (SZ)

Security Architecture	**SZ Internal Vulnerabilities**	**SZ External Vulnerabilities**	**Software and Hardware Authentication**
Standard single software kernel	× No normal zone (NZ) and secure zone (SZ) separation. × Kernel complexity exceeds current formal analysis capabilities and may have exploitable bugs.	× Network access for reporting and updates.	× Relies on protocols and perimeter security techniques requiring software patches
Trusted execution environment such as ARM TrustZone provides a NZ and a SZ	× SZ kernel complexity still exceeds formal analysis capabilities and may have exploitable bugs. × Software needed to switch processor(s) between NZ and SZ.	√ SZ software is isolated from NZ software. × Some SZ hardware resources such as processors are shared with NZ hardware resources.	√ Secure boot of SZ kernel. √ NZ versus SZ state awareness can extend to bus peripherals such as memory and input/output (I/O) controllers, and custom IP.
TECEP adds a SZ with application-specific, autonomic hardware and software	√ Formal analysis of SZ's trusted, application-specific, hardware monitor.	√ SZ hardware and software are fully isolated from NZ hardware and software.	√ Cannot remotely update SZ hardware. √ Prediction capability performs a secure boot load SZ sandbox memory from external flash.

software.

4.3.2 ICS-specific Security Techniques

Security solutions proposed for general embedded platforms are usually not optimized for ICS applications [2]. Because many plants are physically secure, intra-plant communication integrity (to thwart false data injection on sensors or false command injection on actuators) is not the sole concern. Existing reliability analysis can use redundancy to mitigate faults occurring in sensors and actuators. If malicious sensor jamming is a possibility, Cárdenas et al. apply anomaly-based intrusion detection theory for computer systems and networks [3]. However, the ultimate objective of ICS malware is PLC behavior modification from process-harming MTU commands or updates to control, kernel, and driver code. Recent ICS-specific protection schemes seek to close these attack vectors by replicating SCADA network gateways [5] and MTUs [14].

Alternatively run-time monitoring software could attempt to determine if an attack has been successful, much like a vigilant human operator. However, these techniques are generally reactive since they detect attacks beginning in the past or present. The problem with reactive methods is that the plant is already affected, and corrective action must be taken to restore equilibrium. There is a point of no return, formalized by the control-Lyapunov function, beyond which the system becomes unstable. These techniques typically observe either plant reactions to new controller inputs, or controller responses to new sensor measurements. An example of the former was developed by Sha [13]. In this architecture, sensor measurements are monitored by decision logic that determines if a process violation has occurred, as illustrated by Fig. 4.1. If a violation is detected, the decision logic switches control to a high-assurance controller until the system is stabilized. Dai et al. described a fault detection architecture based on observing controller responses to new sensor inputs [6]. Plant measurements are sent to both the production controller and a trusted benchmark version of the controller algorithm. A controller fault is determined by computing the residual of responses of both controllers, as shown in Fig. 4.2. Unfortunately, in either architecture the plant is already affected by the time the fault is detected and interventions are applied.

What we instead propose is an active defense that cannot be disabled by any MTU command or PLC software update, is transparent to the control system designer, and can anticipate controller behavior and plant state for a short period into the future. Section 4.4 describes

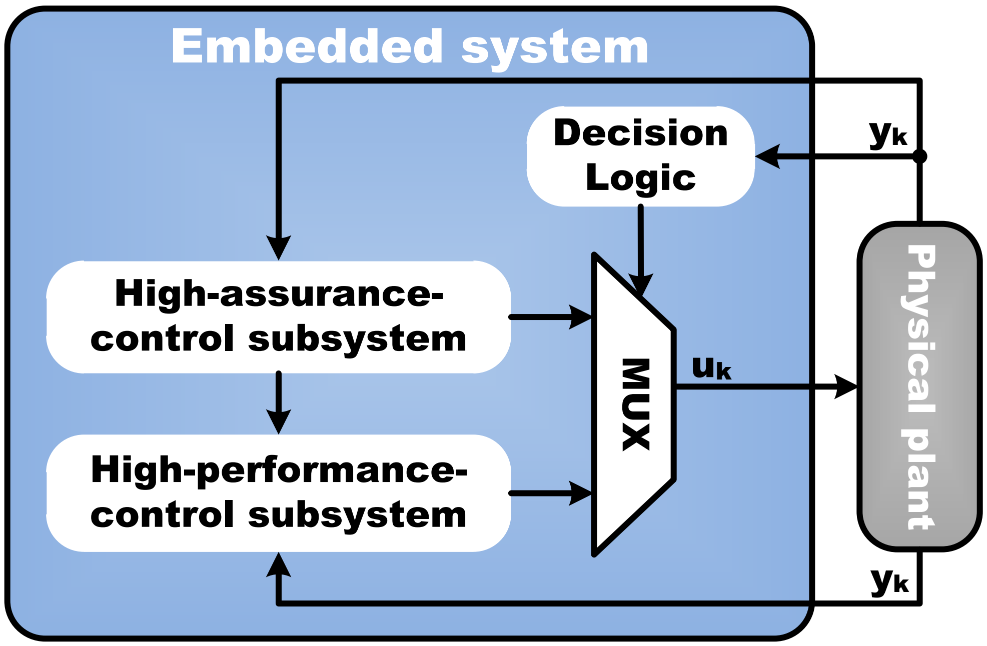

Embedded system
Decision Logic
High-assurance-control subsystem
High-performance-control subsystem
MUX
u_k
y_k
y_k
Physical plant

Figure 4.1: Plant fault detection [13]

how this is accomplished using high-level and interface synthesis, C code formal analysis, hard and soft processors, and configurable logic targeting a configurable system-on-chip platform.

4.4 TECEP Overview

Control system engineers commonly use the model-based design steps outlined with transparent boxes in Fig. 4.3. System specifications are identified, followed by modeling of the plant and control algorithm. Simulation checks that the controller keeps the plant in a stable state within operating specifications. Extensions to modeling environments such as Simulink can automatically generate C code optimized for a particular target processor architecture. System-on-chip platforms containing microcontrollers and FPGA fabric are an appealing target for high-performance controllers because functions may be mapped to either software or hardware. This permits the allocation of independent computational and memory resources to each controller in order to maintain fixed response times, rather than timeshare processors competing for the same memory bandwidth. The post-implementation simulation checks that TECEP additions do not change the original behavior. Component generation

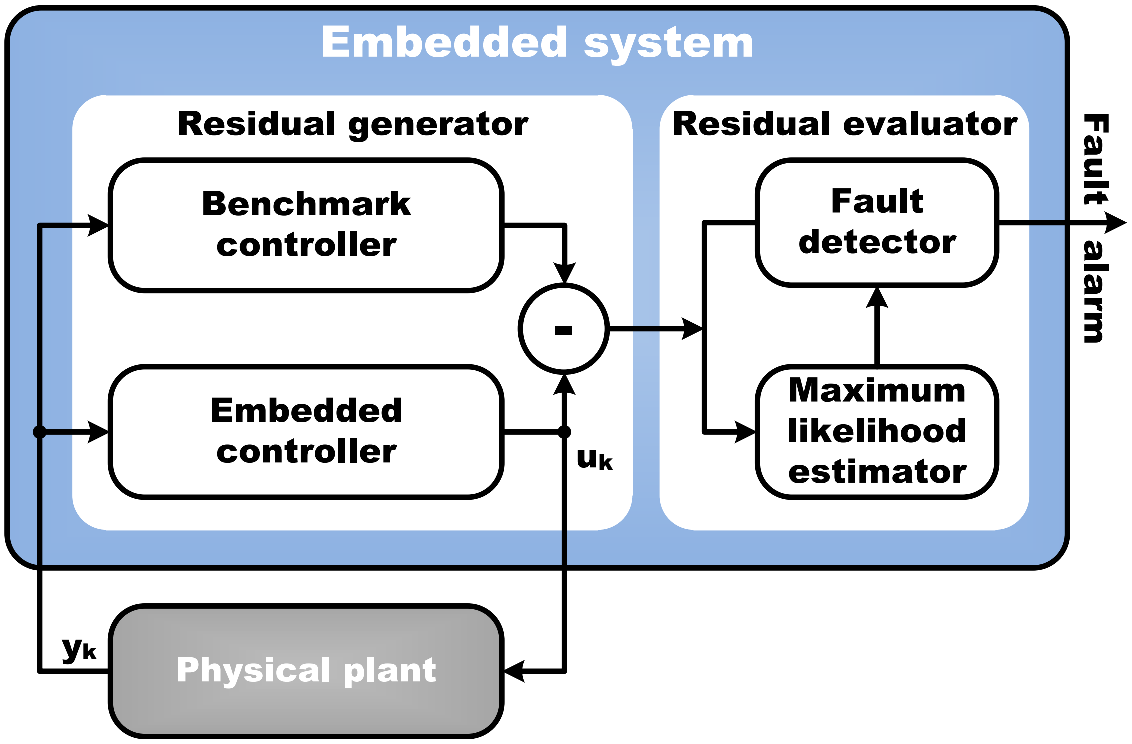


Figure 4.2: Controller fault detection [6]

and integration are discussed in Sections 4.4.1 and 4.4.3.

The extra steps added by TECEP are highlighted in Fig. 4.3 with translucent boxes. A relatively small amount of independent system monitoring code is concerned only with meeting the operating specifications, and is checked with a rigorous software verification framework. Specifics are discussed in Section 4.4.2. The analysis tools require familiarity with formal methods, but do not require hardware verification knowledge even though the monitoring code is ultimately rendered in hardware. Hence the flow separates application, platform, and formal analysis to allow these tasks to be performed by different specialists, with the ultimate goal being semi-automatic synthesis and validation of the additional components.

4.4.1 Platform Components

The system overview shown in Fig. 4.4 includes two software-implemented blocks (production controller code running on both hard and soft processors) and two hardware-targeted blocks (a *hardware monitor* synthesized from formally analyzed C code, and a *junction box*

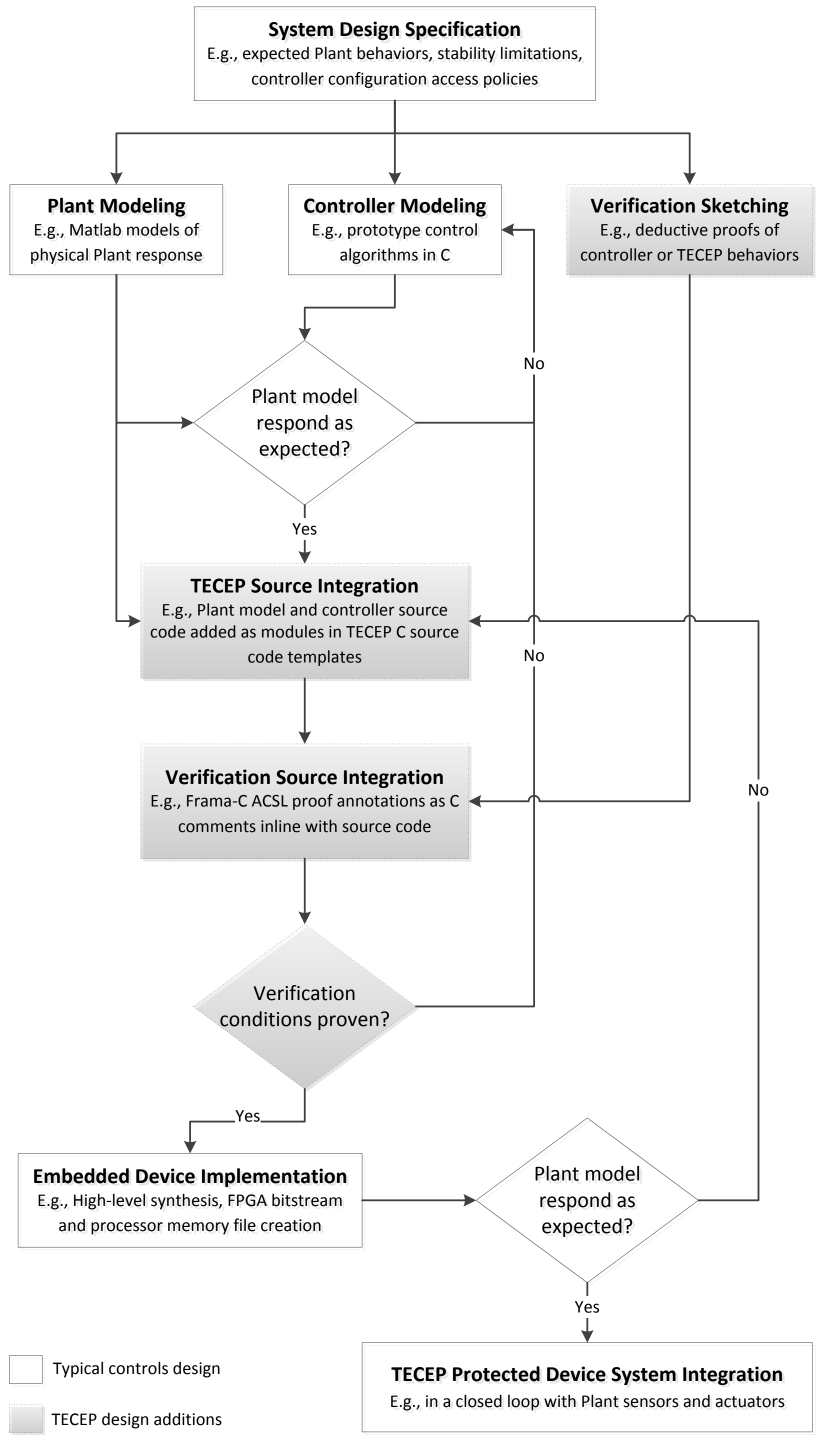

System Design Specification
E.g., expected Plant behaviors, stability limitations, controller configuration access policies
Plant Modeling
E.g., Matlab models of physical Plant response
Controller Modeling
E.g., prototype control algorithms in C
Verification Sketching
E.g., deductive proofs of controller or TECEP behaviors
Plant model respond as expected?
No
Yes
TECEP Source Integration
E.g., Plant model and controller source code added as modules in TECEP C source code templates
No
Verification Source Integration
E.g., Frama-C ACSL proof annotations as C comments inline with source code
No
Verification conditions proven?
Yes
Embedded Device Implementation
E.g., High-level synthesis, FPGA bitstream and processor memory file creation
Plant model respond as expected?
Yes
TECEP Protected Device System Integration
E.g., in a closed loop with Plant sensors and actuators
Typical controls design
TECEP design additions

Figure 4.3: TECEP design flow

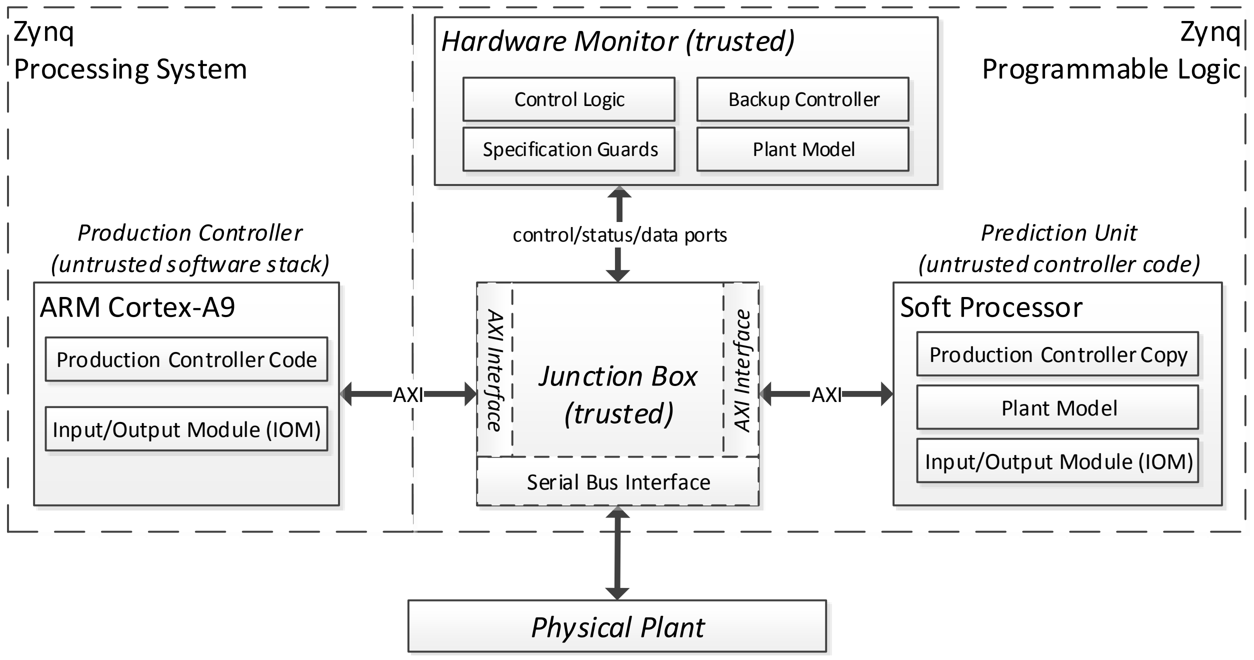


Figure 4.4: Platform components

captured and validated in a hardware description language). Both hardware blocks are invisible to the software blocks, even at the OS driver level. Because the FPGA fabric is not dynamically configured, software or network access to programmable logic configuration ports are disabled. Changes to the programmable logic could require physical access to a secure plant, and are needed only if the process specifications, plant model, backup controller, or switchover policies change. Routine software updates, including production control code revisions, would be stored on network-accessible, external flash memory, and loaded into RAM after a reset.

Production Controller

Implemented in software functions running on the ARM Cortex-A9 processor present on the Zynq-7000, the production controller sends and receives data through an I/O module (IOM), which in turn interacts with the programmable fabric through an AXI bus. Using a real-time kernel such as FreeRTOS, the production controller and IOM are tasks managed by the kernel. Real-time guarantees are generally needed by process control applications.

Hardware Monitor / Backup Controller

The hardware monitor is implemented in the FPGA fabric of the Zynq-7000. Inputs include the $u(t)$ output of the production controller, the $y(t)$ output of the physical plant, the output of the prediction unit, and a status code. Using C code rather than a hardware description language (HDL) allows the hardware monitor to easily integrate a stability-preserving backup controller and the plant model. A specification guard tests whether the outputs of the plant model, physical plant, and prediction unit are within an acceptable range. If any of these values are outside specifications or if the status code indicates an error, the output from the backup controller overrides the production controller's output.

Production to backup controller transitions may be reversed if the production controller fault was merely temporary. This feature is useful in the event of a denial-of-service attack wherein the objective is simply to degrade performance. A probation period ignores the production controller's output for a predetermined number of cycles after a fault is detected even if the controller's output returns to an acceptable value. Hence, the probation period ensures that the production controller is not reinstated too quickly.

It is also important that the backup controller not be invoked at the slightest disturbance, with one implementation defining an upper bound on the number of small disturbances within a given time period. Noise in the system, which may cause false error reporting, will be greatly reduced with the future addition of a Kalman filter. Fault tolerance and any return to the production controller are left to the system specification as requirements differ among various applications.

Prediction Unit

A copy of the production controller and a model of the plant comprise the prediction unit implemented on a MicroBlaze soft processor in the programmable fabric of the Zynq-7000. We ultimately plan to use a soft-core version of the ARM Cortex-M1 processor for Xilinx FPGAs [2] since the production controller may only be available as object code. Since timing characteristics need not be preserved, other options are ARM-to-MicroBlaze binary code translation or ARM emulation on the MicroBlaze. The Zynq-7000's second ARM Cortex-A9 core is not used because it may be needed in multi-threaded control applications. In addition, using an ARM core for prediction requires careful memory and I/O separation from the production core in order to minimize trust assumptions.

Controller and plant functions send and receive data across an AXI bus through an IOM. The production controller IOM forwards MTU commands to the prediction unit IOM, although this link is not yet implemented. FreeRTOS may also be ported to the MicroBlaze/Cortex-M1 in order to more closely mirror the Cortex-A9, with the production controller, plant model and IOMs implemented as tasks managed by the kernel. This will also enable the production Cortex-A9's complete software stack to be executed on the soft processor, allowing the prediction unit to preview attacks that corrupt control algorithm scheduling or driver-level I/O.

Four modes of operation are available in the prediction unit: *Save*, *Restore*, *Normal* and *Accelerate*. *Save* stores the current state of both the plant and the controller, while *Restore* overwrites the current state with the last saved state. *Normal* runs the closed loop system of the plant and controller for one cycle, and *Accelerate* runs the closed loop for a predefined number of cycles. The specific number of prediction cycles used is left to the application. Once the IOM on the soft processor receives a `start` signal from the junction box, the closed loop is run in *Normal* mode. The current state is then saved, and the closed loop is run in *Accelerate* mode. Upon completion, the state of the controller and plant before acceleration is restored, and the accelerated plant model's output is passed through the IOM to the AXI bus.

Junction Box

The production controller, physical plant, prediction unit, and hardware monitor are connected via a junction box which contains all connections between modules in the system, manages the system's flow of control, and scrutinizes all external transfers to and from the physical plant. ARM Cortex-A9 connections to the junction box include AXI interfaces for the production controller's input and output, while soft processor connections consist of AXI interfaces for handshaking and the prediction unit's output. The hardware monitor interfaces to the junction box using simple 32-bit data ports for inputs from the production controller, plant, and prediction unit, as well as the hardware monitor's output and handshaking signals. Two watchdog timers in the junction box monitor the response time of both the production controller and prediction unit; if either unit fails to respond, the hardware monitor is notified with the appropriate status code. The junction box is captured with

HDL code that mostly defines connections and is independent of the production and backup controllers, plant model, and operating specifications. This simplicity makes verification straightforward using established hardware analysis techniques such as model checking.

4.4.2 Hardware Monitor Formal Analysis

Formal analysis tools are incorporated into the high-level design flow to verify functional and security specifications by evaluating mathematical proofs of design code semantic properties. We verify the PID code and TECEP additions using Frama-C, an open source, modular static analysis framework developed specifically for the C language [9]. The framework enables collaboration between various static analysis techniques implemented as plug-ins which can share information. Desired behaviors and other annotations to analyze are captured in the ANSI/ISO-C Specification Language (ACSL). Annotations can specify preconditions and postconditions of a function, predicates, lemmas, axioms, and other assertions and custom logic functions [4]. Formal analysis flows are often complicated by the need to translate or abstract the code under test. However, in this scenario formal tools are easily leveraged as proof annotations are added directly to the design source code. The modular framework also enables verification of isolated functions. This is useful in our security scheme where in order to prove properties of system security the verification space is reduced to only TECEP additions.

ACSL is added directly to the code to be verified and is written as C comments so as not to interfere with standard compilation or HLS tools. These annotations can, however, modify Frama-C's interpretation of function behaviors and variables if they are specifically written to do so. The semantics of ACSL logic expressions are based on mathematical first-order logic, which eases translation of conventional proof languages into proof code. A special type of ACSL annotation, called *ghost code*, is only evaluated by Frama-C and can be independent of any function contract. Ghost code is typically used to specify variable, logic, or functions that are outside of or not related directly to the code under analysis, but are useful in building up proof annotations. Ghost code can also be used within a function to overwrite variables or capture their values for analysis outside of the function. However, using ghost code to interfere with regular program code must be done with care as it can result in inaccurate proofs.

ACSL annotations are verified using various plug-ins within the Frama-C framework in order to offer a variety of provers and analysis techniques. Frama-C can be used to verify the values returned by the hardware monitor when a specification guard detects an out-of-specification condition, and to validate the behavior of modules called by the hardware monitor. Here we use the Jessie plug-in to ensure that the hardware monitor result is driven by the appropriate controller module under all possible conditions. Jessie is a Hoare logic-based plug-in used to prove functional properties via deductive verification [3]. Jessie automatically translates ACSL annotations into verification conditions in the Why language, which can then be submitted to external automatic theorem provers such as Simplify, Alt-Ergo, Z3, Yices, and CVC3. Interactive theorem provers or proof assistants can also be used, such as Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, and Mizar. Frama-C's use of multiple provers combines the strengths of different provers, while time limits cope with undecidability.

4.4.3 High-level and Interface Synthesis

Hardware Monitor

After formal verification, the hardware monitor's software-defined functions are implemented in the programmable fabric using HLS. The `set_directive_allocation` command is applied to the C-synthesis process to restrict the number of floating-point cores generated, and the `config_bind` command reduces the resources used by those instantiated cores. An `ap_none` interface is added to each of the inputs to create simple data ports with no additional handshaking signals. An `ap_ctrl_hs` interface added to the top level function provides basic handshaking signals such as `start` and `done` for the operation of the hardware monitor. No AXI slave adapters are needed, as the hardware monitor is connected directly to the junction box using simple 32-bit data ports. Once HLS and interface synthesis processes are complete, the hardware monitor is exported as an IP block for use in Vivado Design Suite.

AXI Interconnects

The junction box itself uses simple 32-bit input and output ports; however, the ARM and soft processors require an AXI interface for sending and receiving data. Vivado Design Suite has no simple means of adding an AXI slave adapter to a simple 32-bit data port, but Vivado HLS does. A trivial C function that returns its input argument is used to create the interface adapter. Depending on the direction of the data transfer, an AXI4-Lite slave adapter is added to either the input argument or the return port. The `ap_none` interface protocol is added to each of the interface adapters with two exceptions: the production controller's output and the prediction unit's output use the `ap_ctrl_hs` protocol. Each interface module utilizes on average 74 flip-flops and 20 lookup tables, though there is some variation based on the data type being used in the C function. After completion of HLS and interface synthesis, the interface adapters are also exported as IP blocks for use in Vivado Design Suite.

4.4.4 Limitations and Tradeoffs

There are always cost concerns arising from platform and development flow complications. For example, Frama-C requires significant manual intervention and expertise. However the TECEP additions are independent in a trust sense from the base ICS architecture and code, and offer a lower cost, on-chip, digital logic alternative to classical mechanical interlocks used in safety critical environments. A configurable SoC's cost premium over a standard microcontroller may be eclipsed by the value of the plant.

The plant state preview window is ultimately limited by processing power and the need to periodically synchonize the model with the physical plant to keep up with changes due to disturbances or commands. Prediction may be omitted if process state also depends on events outside the model.

While the inability to remotely update TECEP logic is a security asset, the need for PLC access to modify the model, process specifications, and backup controller incurs a higher maintenance cost. We are investigating a remote update protocol permitting range-limited, authenticated changes to certain system parameters in order to adjust for effects such as aging.

4.5 Motor Controller Example

A simple motor controller example described in [3] is used to test and demonstrate the security features described in this work. For this example, the production controller is a proportional-integral-derivative controller while the backup controller is a proportional-only controller with a fixed reference input. The plant is a motor emulated in software functions on the ARM. This closed-loop system is run once per millisecond.

4.5.1 Control Flow

The system's flow of control is event-driven and is shown in Fig. 4.5. Each cycle begins after a 1 ms delay dictated by an AXI Timer interfaced with the ARM. During the delay the production controller, prediction unit and hardware monitor lie dormant. Although the physical plant will operate on its own timing, the currently emulated plant begins operation once the ARM recognizes 1 ms has elapsed. The emulated plant function reads its input from the IOM, processes it, and sends its output to the IOM. The plant's I/O is simply passed through the junction box, with no handshaking and non-blocking reads and writes. When the plant's process is complete, the production controller similarly reads its input from the IOM, processes it, and writes the output to the IOM. The HLS interface module between the junction box and the production controller performs a blocking read on the controller's output. A new value is passed to the junction box, which signals the MicroBlaze prediction unit to begin, and a blocking read is then performed on the prediction unit's output. When the prediction unit finishes, the junction box sends the outputs of the production controller, physical plant, and prediction unit to the hardware monitor and signals it to begin operation. The hardware monitor's output is written to the physical plant, and the system lies dormant until the next cycle begins.

The junction box's watchdog timers monitor the response time of the production controller and the prediction unit. If either unit fails to respond before its timer expires, a corresponding status code is sent to the hardware monitor. Because security needs vary among control systems, the hardware monitor's reaction to various status codes is not fixed. A zero status code implies normal operation, and any non-zero code results in a transfer of control to the backup controller. The backup controller is then invoked each cycle based on the junction box's internal timer rather than upon the completion of the production controller and prediction unit.

4.5.2 Application-specific Attributes

Some aspects of the implementation process are specific to the motor controller example and will differ among various control systems. One such aspect is the hardware monitor's response to the loss of the production controller or prediction unit. In some systems, the production controller may be allowed to recover, or the loss of prediction may not be considered critical.

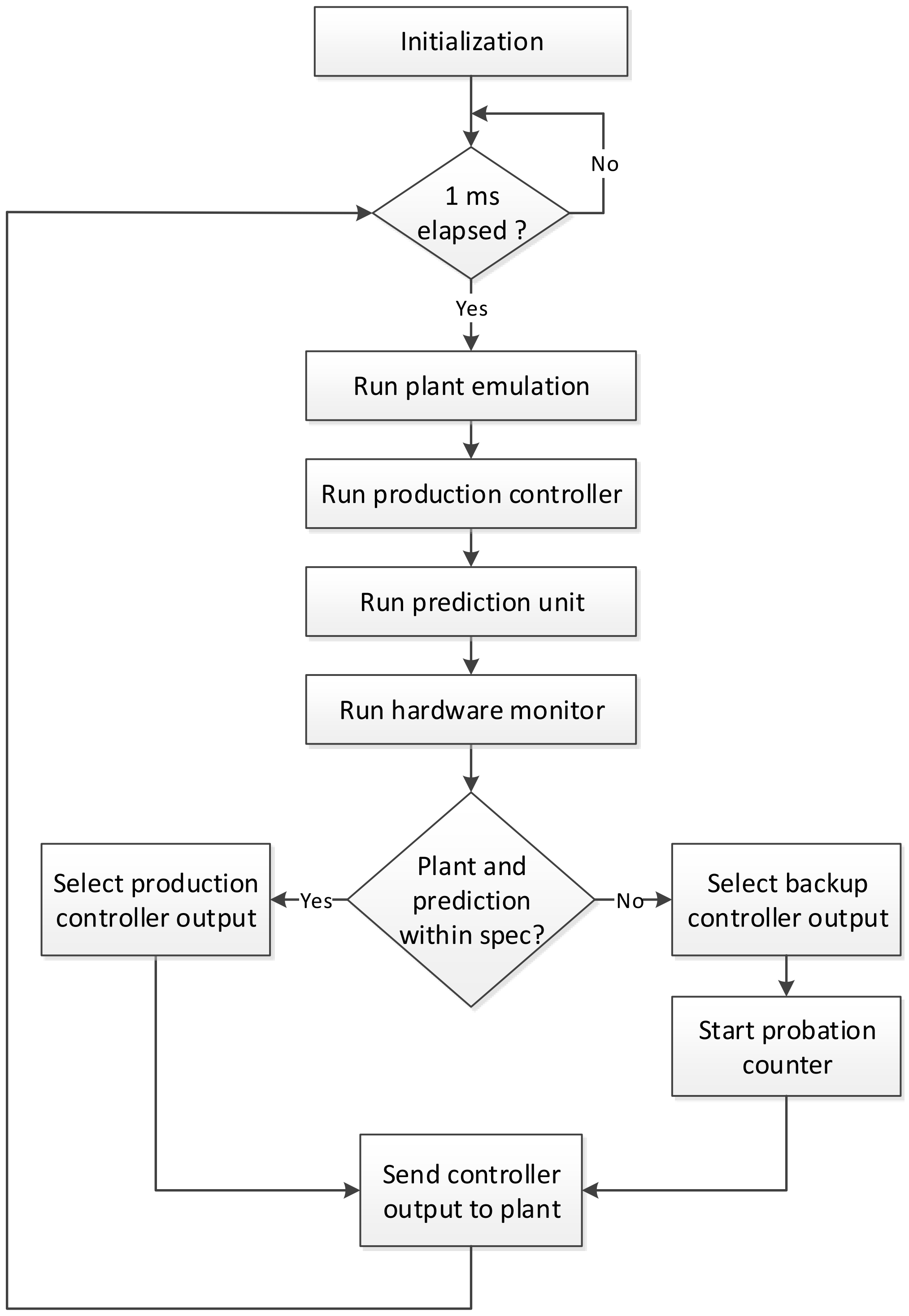

Initialization
No
1 ms elapsed ?
Yes
Run plant emulation
Run production controller
Run prediction unit
Run hardware monitor
Select production controller output
Yes
Plant and prediction within spec?
No
Select backup controller output
Start probation counter
Send controller output to plant

Figure 4.5: Control system event sequence

```
//@ ghost float ghost_y_model = 0.0f;
//@ ghost int   ghost_backup = 0;
//@ ghost float ghost_u_hw = 0.0f;
/*@
   assigns \nothing;
   behavior verify_all_valid:
      assumes y_physical >= y_min     &&  y_physical <= y_max    &&
              ghost_y_model >= y_min  &&  ghost_y_model <= y_max &&
              y_accel >= y_min        &&  y_accel <= y_max;
      ensures ghost_backup == 0;
   behavior verify_any_invalid:
      assumes y_physical < y_min      ||  y_physical > y_max     ||
              ghost_y_model < y_min   ||  ghost_y_model > y_max  ||
              y_accel < y_min         ||  y_accel > y_max;
      ensures \result == ghost_u_hw;
   disjoint behaviors;
   complete behaviors;
*/

#define BACKUP_HOLD_COUNT 200
float hw_monitor(float u_sw, float y_physical, float y_accel)
{
   static int   backup_hold;
   static float y_model;

   if (reset)
      y_model = hw_plant_model(u_sw);

   float u_hw = hw_controller(y_physical);
   //@ ghost y_model = ghost_y_model;  // Assignment here allows
     y_model reset
   int backup = !hw_spec_guard(y_physical) ||
                !hw_spec_guard(y_model)    ||
                !hw_spec_guard(y_accel);

   backup_hold = (backup) ? (BACKUP_HOLD_COUNT) : (backup_hold - 1);
   backup_hold = (backup_hold < 0) ? 0 : backup_hold;
   float u = (!backup && !backup_hold) ? u_sw : u_hw;
   y_model = hw_plant_model(u);
   //@ ghost ghost_backup = backup;
   //@ ghost ghost_u_hw = u_hw;
   return u;
}
```

Figure 4.6: `hw_monitor` function annotated with Frama-C

In this example, if the junction box determines that either unit has failed, the invocation of the backup controller is permanent until the system is reset. The prediction unit's forecast window and the probation counter length are 200 system cycles (i.e. 200 ms), as shown in line 20 of Fig. 4.6. However, these values may be adjusted.

Currently, saving and restoring state in the prediction unit is performed by changing global variables; this method is feasible only if control system source code is available. Running FreeRTOS on the MicroBlaze avoids the need for source code. This new method uses a second production controller and plant model. The primary set runs in *Normal* mode. An administrative task copies the primary set's stack, including state variables, onto the secondary set's stack, which is run in *Accelerate* mode. Alternatively one can clone the primary production controller and plant model tasks, run the cloned tasks in *Accelerate* mode, pass the accelerated output to the IOM task, and delete the cloned task.

4.5.3 Hardware Monitor Code Analysis

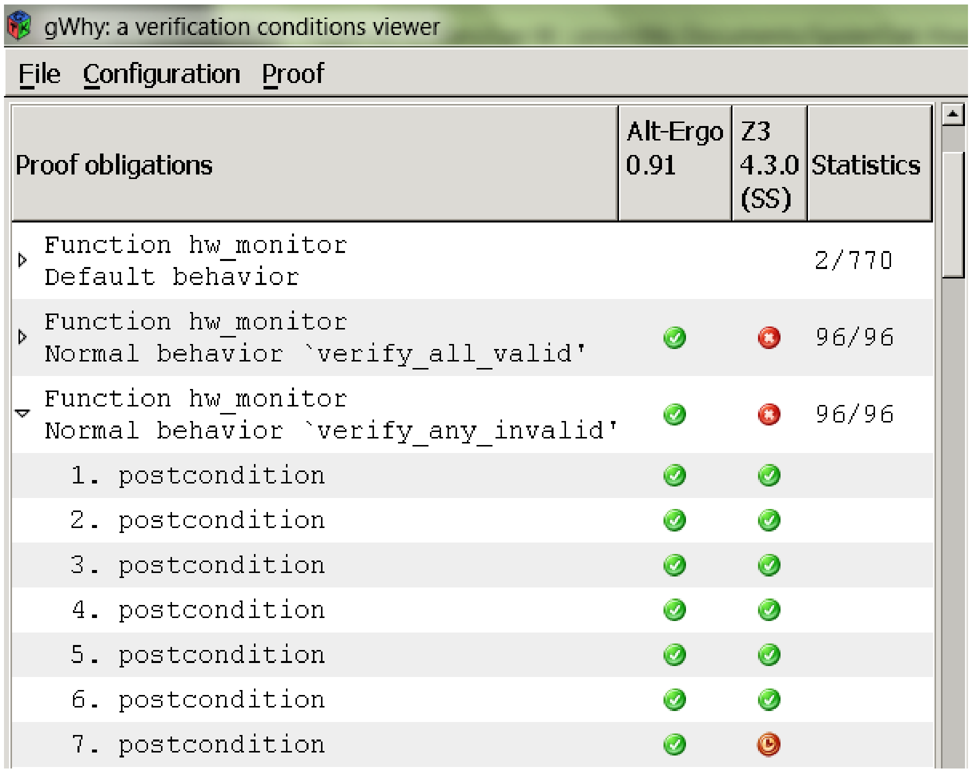
Formal methods can be used at this stage in the design process to confirm TECEP security additions such as the hardware-implemented backup controller module taking over when a process parameter goes out of specification. To accomplish this we analyze the TECEP hardware monitor module, which calls the controller prediction, specification guard, and backup controller modules. The hardware monitor source code and Frama-C annotations are provided in Fig. 4.6. A combination of ghost code and function contracts are used to reason about hardware monitor behaviors under valid and invalid specification conditions. The `assigns` clause on line 5 simply specifies that the `hw_monitor` function does not have any side effects (i.e., does not assign any values which are not local). This clause is followed with two function contracts consisting of `behavior` clauses which test various conditions.

The first behavior, `verify_all_valid` starting on line 6 of Fig. 4.6, aims to verify that the `hw_monitor` does not trigger the backup condition whenever the specification guard does not detect an out-of-specification condition. To test this normal operating condition we assume that the plant response, plant model response, and predicted plant model response are all within stability specifications. The `assumes` clause can specify valid `y_physical` and `y_accel` value ranges directly because they are inputs of the function. Value ranges of `y_model`, which is local to the function, can be overridden within the function with a ghost code variable, `ghost_y_model`, on which assumptions can be made in the same manner as function inputs and outputs. Another ghost variable, `ghost_backup` is used to store the value of the local `backup` variable within the function. This enables an `ensures` clause using `ghost_backup` which tests if the backup controller is ever set to active at the completion of the function.

The second behavior, `verify_any_invalid` starting on line 11 of Fig. 4.6, seeks to prove that the `hw_monitor` will always choose the backup, hardware-implemented controller response when any unit is out of specification. For this behavior we assume that either `y_physical`, `y_accel`, or `y_model` has a value below `y_min` or above `y_max`. Under this assumption the hardware specification guard should detect an out-of-specification condition and switch control to the hardware implemented backup controller, `u_hw`. A ghost variable, `ghost_u_hw`, is used to capture the local value of `u_hw` at the end of the function. The `ensures` clause on line 15 tests that the function returns a value which equals this captured value. Lines 16 and 17 test that this and the previous behavior form a set of proofs that are both disjoint and complete, respectively. The set of behaviors is considered `disjoint` when they do not overlap and `complete` when all cases are covered.

The Jessie to Why translation for the `verify_all_valid` and `verify_any_invalid` function contracts each results in 96 total postcondition verification conditions (or proof obligations) representing various possibilites for input values and corresponding branches of the function that could be executed, as seen in Fig. 4.7. Two additional proof obligations of interest are also created to check if our set of behaviors are disjoint and complete. Other obligations that were generated for the `hw_monitor` function's default behavior and safety were not considered in this analysis. The simplicity of the analyzed code and its hardware implementation as a sequential state machine reduce the likelihood of semantic mismatches between Frama-C and HLS.

The results of running two proof checkers are also shown in Fig. 4.7. All conditions must be discharged for the behaviors to be proven valid, which is accomplished using the Alt-Ergo prover within only a few minutes. We used a 10 second timeout threshold for this experiment. The Z3 prover was unable to prove some of the postconditions before hitting the timeout limit, e.g., Postcondition 7 in Fig. 4.7. Increasing the timeout limit could possibly enable the Z3 prover to discharge this verification condition. However, this is unnecessary as these problematic proof obligations have been discharged by the other prover. The two obligations checked in the default behavior section prove that the set of behaviors are both disjoint and complete. Thus, Frama-C is able to provide confidence that our TECEP protections both correctly select the production controller output under normal operating conditions and select the backup controller in the event of malicious production controller behaviors or anomalous plant sensor data.



Proof obligations	Alt-Ergo 0.91	Z3 4.3.0 (SS)	Statistics
Function hw_monitor Default behavior			2/770
Function hw_monitor Normal behavior `verify_all_valid'	✓	✗	96/96
Function hw_monitor Normal behavior `verify_any_invalid'	✓	✗	96/96
1. postcondition	✓	✓	
2. postcondition	✓	✓	
3. postcondition	✓	✓	
4. postcondition	✓	✓	
5. postcondition	✓	✓	
6. postcondition	✓	✓	
7. postcondition	✓	◷	

Figure 4.7: Jessie deductive verification results

4.5.4 Module Integration

The system is implemented with Vivado Design Suite 2013.2 on a 64-bit Linux workstation host running the 2.6.32-28-generic kernel on an 4-core, 2.8 GHz Intel Core i7 processor with 24GB of RAM. A Vivado Design Suite project is used to create the junction box using HDL files and to export it in IP block format. This IP block, along with the hardware monitor and interface adapter IP blocks generated by Vivado HLS, are imported into the IP catalog of a new Vivado Design Suite project.

A block diagram is created with the Zynq processing system and a performance-optimized MicroBlaze soft processor with 16KB of local memory. Three interface adapters (the production controller's input and output, and the emulated plant's I/O) are added as AXI peripherals to the ARM. Two interface adapters (the prediction unit's output, and handshaking signals) are added as AXI peripherals to the MicroBlaze. The hardware monitor is instantiated in the system and connected to the junction box, as are each of the interface adapters. After implementation, a bitstream is exported to the Xilinx SDK, wherein all software is compiled and downloaded to the ARM and MicroBlaze processors.

4.5.5 System Behavior

In this example, latent malicious behavior is inserted into the production controller. The system begins with a unit step response followed by activation of the latent malware. This malware attempts to drive the plant's output to its maximum (clipped) value beginning at $t = 350$ ms. With no countermeasures in place, Fig. 4.8 shows the plant's output exceeding the safe limit of 3.2. Plant behavior with the hardware monitor appears in Fig. 4.9. Without prediction, the plant's output approaches the acceptable limit at $t = 480$ ms before being corrected by the backup controller. With prediction, the hardware monitor proactively invokes the backup controller at $t = 280$ ms, thereby preventing the physical plant from reaching an unsafe state. The backup controller will remain active until the system is reset. Fig. 4.10 shows the plant's output with prediction and automatic resumption of the production controller. As in Fig. 4.9, the prediction unit forecasts the future consequences of the malware, and the hardware monitor again preemptively switches to the backup controller at $t = 280$ ms. The hardware monitor's probation counter expires and the production controller is reinstated when the malware ends at $t = 650$ ms since 200 ms before this time the physical plant, plant model, and predicted plant state are found to be within specifications. General mechanisms are being developed to manage transitions between controllers in order to avoid excessive overshoot caused by stale state information.

4.5.6 Time and Resource Utilization

Despite the implementation of software functions in hardware, performance is not an objective in this work, as the system needs to operate only once per millisecond. The ARM processor operates at 667 MHz; the MicroBlaze processor, along with the remaining hardware, operates at 140 MHz. Both processors utilize full optimization and hardware floating point instructions. One iteration of the prediction unit requires 1.43 microseconds to complete. Considering this runtime, the comparatively minimal runtimes of the production

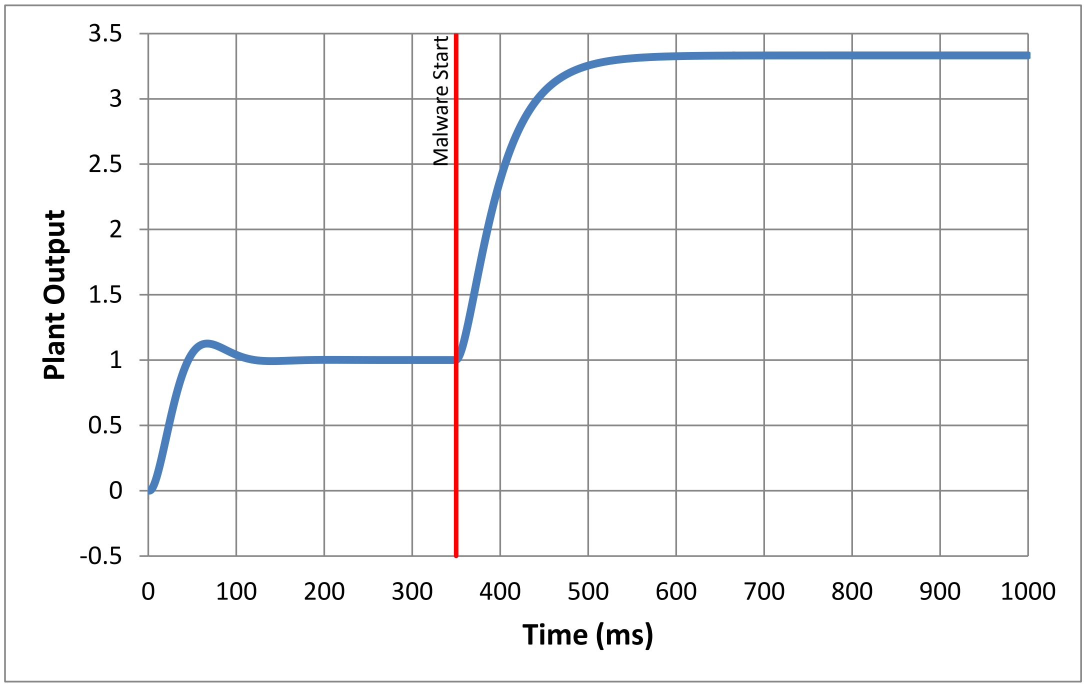


Figure 4.8: Unprotected plant behavior

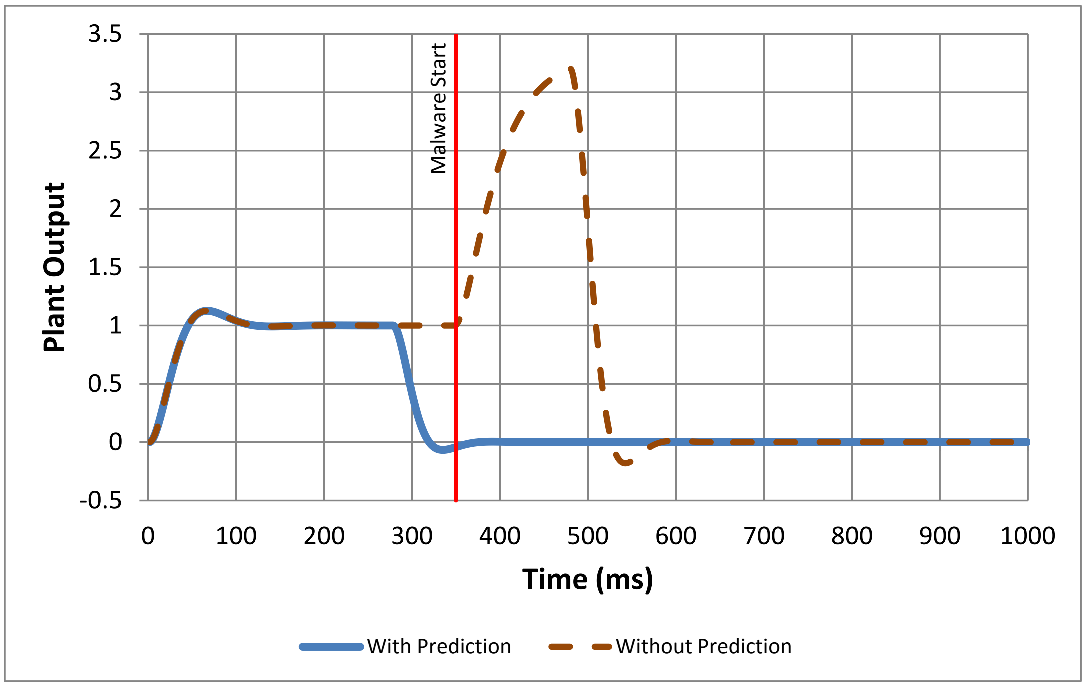


Figure 4.9: Protected plant behavior with and without prediction

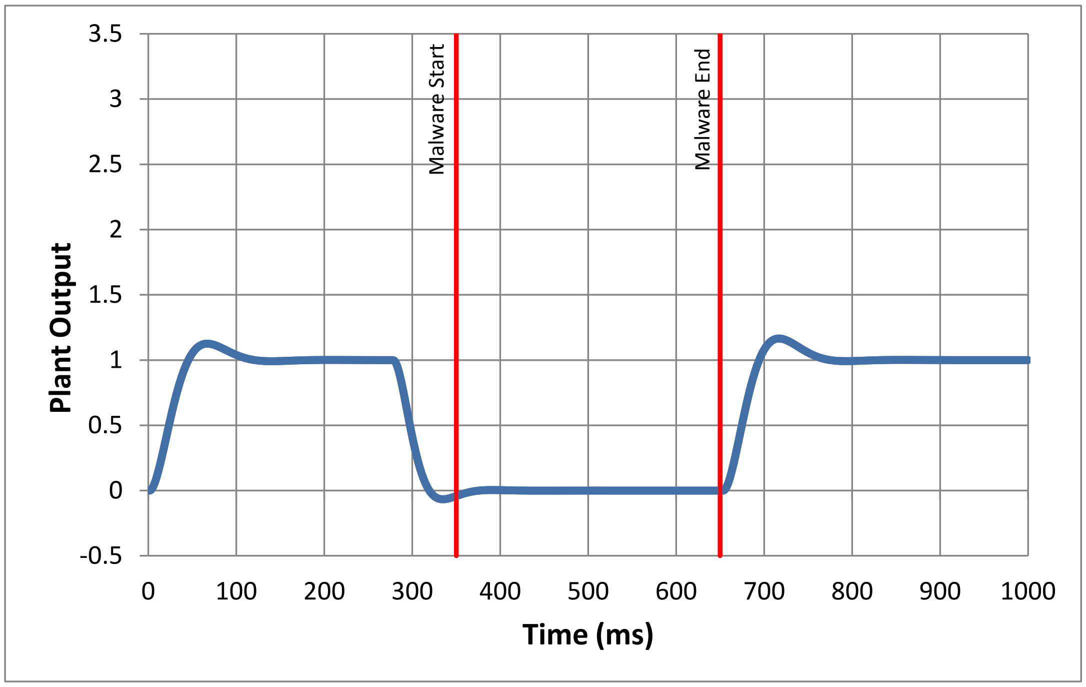


Figure 4.10: Protected plant behavior with return to the production controller

controller and hardware monitor and all communication overheads, it is possible to predict over 500 cycles into the future during each 1 ms system cycle. This prediction window is currently limited by the MicroBlaze's computational throughput. If a larger prediction window is required, more intensive optimization of the MicroBlaze or a faster platform speed grade may be necessary.

The resources consumed by the components and the overall system are shown in Table 4.2. The prediction unit's MicroBlaze processor has a large resource usage because it is configured with a five-stage pipeline, hardware support for floating point addition and multiplication, and an AXI timer peripheral for measuring code latency.

4.6 Conclusions and Current Work

Existing approaches to control system security add generic hardware or software layers to help isolate and secure applications. We instead use rigorous verification of application-specific hardware to counter software reconfiguration attacks on critical processes, accomplished

Table 4.2: Zynq-7020 programmable logic resource usage

	FF	**LUT**	**DSP**	**BRAM**
Hardware Monitor	677	1046	5	0
HLS interfaces	295	78	0	0
Junction Box	70	80	0	0
Prediction Unit	2813	3174	2	4
Total Used	3855	4378	7	4
Available	53200	106400	140	220
Percent Used	7	4	5	2

through the non-standard use of existing languages, tools, platforms, process specifications and models. C is used for all application-specific software- and hardware-implemented functions in the system-on-chip platform. This conforms with the C code automatically synthesized from model-based design tools or manually generated by control system designers, and enables the novel use of C code static analysis tools for functions implemented in hardware.

The Frama-C analysis tool best suits our needs because of ACSL's first-order logic expressive power for annotating partial functional specifications, support for floating point arithmetic so that fixed point error analysis is not needed, and use of several provers to check the validity of assertions. Although HLS from C is a commercial technology, our additional use of formal specifications expressed in ACSL addresses the equally important matter of high-level verification of hardware functionality, and further unifies hardware and software development. We use a three-level abstraction hierarchy from ACSL (for specifying security-related system properties) to C (for capturing the system implementation without any hardware-level complications) to configurable logic (where hardware utilization can be controlled and reported by the Xilinx synthesis and implementation tools). The lack of aggressive timing goals promotes hardware generation without any manual intervention.

We are presently applying TECEP to an electromechanical physical plant where non-ideal effects such as noise, disturbances, and actuator limitations are present. The Quanser ROTPEN-SE apparatus allows testing our approach on an increasingly complex set of systems, from simple rotary motion control, to an inherently stable gantry crane system, and finally to an inherently unstable inverted pendulum system [20]. Because the ZedBoard is not yet supported by Quanser, we created a custom SPI interface to the physical system as well as custom Simulink blocks to allow the automated implementation of our controllers.

A Kalman filter will serve as a natural way to accurately estimate the controlled system's state for short time horizons. TECEP's ability to scale up to cyber-physical systems and applicability to non-control domains are also being investigated.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number CNS-1222656. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We are grateful for ZedBoard and tool donations from Xilinx, Inc.

References

[1] ARM Ltd. ARM security technology: Building a secure system using TrustZone technology, 2009. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[2] ARM Ltd. Cortex-M1 FPGA processor, 2013.

[3] Daniele Bagni and Duncan Mackay. Floating-point PID Controller Design with Vivado HLS and System Generator for DSP, January 2013. XAPP1163 (v1.0).

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language: Version 1.4 Boron-20100401.* CEA LIST, France Télécom, INRIA Saclay, LRI – Univ Paris-Sud, 2010. http://frama-c.com.

[5] A.N. Bessani, P. Sousa, M. Correia, N.F. Neves, and P. Verissimo. The Crutial way of critical infrastructure protection. *Security Privacy, IEEE*, 6(6):44–51, Nov 2008.

[6] A.A. Cárdenas, S. Amin, and S. Sastry. Secure control: Towards survivable cyber-physical systems. In *Distributed Computing Systems Workshops, 2008. ICDCS '08. 28th International Conference on*, pages 495–500, Jun 2008.

[7] Alvaro A. Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. Attacks against process control systems: risk assessment, detection, and response. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'11, pages 355–366, 2011.

[8] Alvaro A. Cárdenas, Saurabh Amin, and Shankar Sastry. Research challenges for the security of control systems. In *Proceedings of the 3rd Conference on Hot Topics in Security*, pages 6:1–6:6, Berkeley, CA, USA, 2008. USENIX Association.

[9] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in: Fluorine-20130501.* CEA LIST, Software Reliability Laboratory, Saclay, F-91191, 2013. `http://frama-c.com`.

[10] C. Dai, S.H. Yang, and Liansheng Tan. An approach for controller fault detection. In *Fifth World Conference on Intelligent Control and Automation (WCICA)*, volume 2, pages 1637–1641, Jun 2004.

[11] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 9th edition, 2000.

[12] N. Falliere, L. O'Murchu, and E. Chien. W32.stuxnet dossier, 2011.

[13] Kelly Jackson Higgens. Spaf on security, Oct 2013. `http://www.darkreading.com/vulnerability/spaf-on-security/240162511`.

[14] J. Kirsch, S. Goose, Y. Amir, Dong Wei, and P. Skare. Survivable SCADA via intrusion-tolerant replication. *Smart Grid, IEEE Transactions on*, 5(1):60–70, Jan 2014.

[15] Lee W. Lerner, Mohammed M. Farag, and Cameron D. Patterson. Run-time prediction and preemption of configuration attacks on embedded process controllers. In *International Conference on Security of Internet of Things (SecurIT 2012)*, Kerala, India, Aug 2012.

[16] Lee W. Lerner, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson. Using high-level synthesis and formal analysis to predict and preempt attacks on industrial control systems. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 209–212, Monterey, CA, Feb 2014.

[17] Claude Marché and Yannick Moy. *The Jessie plugin for Deductive Verification in Frama-C.* INRIA Saclay - Télé-de-France and LRI, CNRS UMR 8623, 2013. `http://proval.lri.fr/`.

[18] MathWorks. Embedded Coder. `http://www.mathworks.com/products/embedded-coder/`.

[19] Nicole Perlroth and David E. Sanger. Nations buying as hackers sell flaws in computer code. *New York Times*, Jul 13, 2013.

[20] Quanser Inc. The rotary control lab, 2012. `http://www.quanser.com/flippers/Rotary/2012/`.

[21] Dan Rosenberg. Unlocking the Motorola bootloader, Apr 2013. `http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html`.

[22] William Safire. The Farewell Dossier. *New York Times*, Feb 2, 2004.

[23] David E. Sanger. N.S.A. nominee promotes cyberwar units. *New York Times*, March 11, 2014.

[24] Carl Schuett, Jonathan Butts, and Stephen Dunlap. An evaluation of modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 7(1):61 – 68, 2014.

[25] Lui Sha. Using simplicity to control complexity. *Software, IEEE*, 18(4):20–28, Jul–Aug 2001.

[26] Joseph Weiss. *Protecting industrial control systems from electronic threats.* Momentum Press, 2010.

[27] Mark Zeller. Myth or reality – does the Aurora vulnerability pose a risk to my generator? In *Protective Relay Engineers, 2011 64th Annual Conference for*, pages 130–136. IEEE, 2011.

Chapter 5

Run-time Prediction and Preemption of Configuration Attacks on Embedded Process Controllers

5.1 Abstract

Embedded electronics are widely used in cyber-physical process control systems (PCSes), which tightly integrate and coordinate computational and physical elements. PCSes have safety-critical applications, such as the supervisory control and data acquisition (SCADA) systems used in industrial control infrastructure, or the flight control systems used in commercial aircraft. Perimeter security and air gap approaches to preventing malware infiltration of PCSes are challenged by the complexity of modern networked control systems incorporating numerous heterogeneous and updatable components such as standard personal computing platforms, operating systems, and embedded configurable controllers. Global supply chains and third-party hardware components, tools, and software limit the reach of design verification techniques. As a consequence, attacks such as Stuxnet have demonstrated that these systems can be surreptitiously compromised.

We present a run-time method for process control violation prediction that can be leveraged to enhance system security against configuration attacks on embedded controllers. The prediction architecture provides a short-term projection of active controller actions by embedding an accelerated model of the controller and physical process interaction. To maintain convergence with the physical system, the predictor model state is periodically synchronized with the actual physical process state. The predictor is combined with run-time guards in a root-of-trust to detect when the predicted process state violates application specifications. Configurations can be screened before they are applied or monitored at run-time to detect subtle modifications or Trojans with complex activation triggers. Advanced notification of process control violations allows remedial actions leveraging well known, high-assurance techniques, such as temporarily switching control to a stability-preserving backup controller. Experimental simulation results are provided from a root-of-trust developed for an aircraft pitch control system.

5.2 Introduction

A process control system (PCS) is an embedded computer platform used to monitor and control physical processes. PCSes are a subset of cyber-physical systems, which tightly integrate and coordinate computational and physical elements. One example of a PCS is feedback control, where an embedded controller uses sensor measurements of a physical plant to compute feedback signals preserving system stability. PCSes are widely used in safety-critical infrastructure applications such as power grids, assembly lines, water systems, pipelines, power plants, and other industrial systems [1, 5]. Recent PCS attacks such as Stuxnet, which is described as the real start of cyber warfare, have highlighted embedded system vulnerabilities and the inadequacy of existing security solutions.

The Stuxnet worm infects Windows computers, spreads via networks and removable storage devices, and exploits four zero-day attacks (previously unknown vulnerabilities). Antivirus software missed the attack because programmable logic controller (PLC) rootkits hide Stuxnet modifications to the system, and two stolen certificates validate new drivers. The goal of Stuxnet is to sabotage a specific physical system by reprogramming embedded controllers to operate outside their nominal bounds by intercepting routines that read, write, and locate PLC commands and data. Many security companies state that Stuxnet is the most sophisticated attack they have ever analyzed [3], and it is estimated to have infected 50,000–100,000 computers. The primary target is believed to be the Bushehr nuclear plant in Iran, and likely caused a 15% drop in production of highly enriched uranium [4].

PCSes are usually assembled from commercial-off-the-shelf (COTS) components and third-party intellectual property (IP). Despite the lack of trust in such components, feasible alternatives may not exist for the timely development of new systems. Trojans can be introduced into the global supply chain as either hardware or software modifications to embedded components. Since controllers are often programmable, many Trojans need only be implemented in software. Even hardware trust may be misplaced in configurable platforms, such as those utilizing field programmable gate arrays (FPGAs). PCS threats can originate from numerous sources, including hostile governments, terrorist groups, disgruntled employees, malicious intruders, and untrusted insiders.

Embedded system security solutions can be classified as either design-time or run-time techniques [15]. Design-time approaches typically seek to verify that a system is error-free pre-deployment. An example method is formal verification of system implementation to design specifications. However, system-level verification is difficult to achieve for complex assemblies of heterogeneous components. Such design-time techniques are expensive in terms of both time and effort, and can be only afforded for a limited set of applications.

Run-time techniques add trusted components to provide assurances about certain aspects of system behavior. An example of a trust anchor component is a trusted platform module (TPM) commonly used in modern personal computers [10]. Common additions include encryption and authentication modules that help assure information integrity and provide isolated compartments for applications with various levels of privilege. Such techniques can be costly in terms of design overhead and added latency and thus are often not appropriate for applications such as high-performance or general purpose computing. Yet for many applications, such as embedded cyber-physical systems, the security gains that can be achieved through trusted run-time anchors justify their presence.

We generate run-time components to simultaneously address design-for-security, -trust, and -reliability (DFSTAR). To protect against Stuxnet-like cyber threats, a secure cyber-barrier is placed around the system's control path. System behavior checks are synthesized at design-time from an application's operational and security specifications. Using this methodology, a tailored trustworthy control flow is created for the target application. This fundamentally new approach is not domain-specific and provides a proactive solution for sustaining system-level security with reliable control. Existing verification techniques are complemented but not exclusively relied upon to ensure functional system trust and security compliance. System-level PCS reliability is also addressed by incorporating specifications that should already be defined for high-reliability systems.

Farag et al. presented a configurable hardware-assisted application rule enforcement (CHARE) protection scheme to ensure an embedded system adheres to system specifications [8]. CHARE addresses several aspects of control security, including high-assurance module interactions and configuration programming in embedded systems. A centralized CHARE trust anchor serves as the most privileged root-of-trust for control flow, and inserts a distributed set of policy-aware specification guards on module interfaces. Specification guards provide on-line monitoring and proactive enforcement of policy rules emanating from security, performance, or reliability specifications. CHARE components tailor the hardware surrounding a system's datapath and control logic to the intended application, but do not affect the implementation of the original logic itself. A hardware-oriented solution offers resistance to software attacks and the performance necessary to implement real-time checks. CHARE reconfiguration allows for policy changes, but the trust anchor itself can only be updated with physical access or trusted channels.

The DFSTAR methodology encourages the synthesis of behavioral expectations of an entire system, including physical processes themselves. The models developed during the design stage of a cyber-physical system can be viewed as a manifestation of such system expectations. We propose that a security architecture for PCSes providing secure configuration management can be synthesized from these models following the DFSTAR methodology. In this work, a protection scheme utilizing CHARE extended with a novel process control violation prediction method is developed for embedded PCS controllers. Prediction logic incorporates a second instance of the embedded controller connected to a physical plant model running faster than real time in order to predict the future state of the physical system. The model's state is periodically synchronized with the physical plant's state to prevent state divergence. CHARE specification guards check if future system states will violate system-level policies. Controller configuration updates are tested against the current state of the process before they are applied. Additionally, if a violation is predicted after applying an update, guards switch from the faulty controller to a high-assurance, stability-preserving, backup controller until the system is stabilized. For process control networks, CHARE may be collectively applied over the network of controllers, sensors, and supervisory software.

The remainder of this paper is organized as follows: Section 5.3 surveys existing run-time approaches to embedded PCS reliability and security. Section 5.4 describes the concept of process control violation prediction and how it can be used to defend against configuration attacks on networked PCS controllers. This section also describes a prototype architecture and implementation flow for the protection system. Experimental results from applying the prediction method to a flight pitch controller simulation are provided in Section 5.5. Finally, conclusions and future work are summarized in Section 5.6.

5.3 Background

The need for high-assurance control of physical processes by cyber-systems has led to the development of several fault detection techniques. In the case of embedded controller faults or attacks, erroneous controller behavior is ideally detected and corrected while the physical process can still return to equilibrium. PCS fault detection techniques typically observe either physical process measurements to new controller inputs or controller responses to new sensor measurements. Sha introduced a protection architecture based on monitoring physical process measurements to detect faults [13]. In this architecture, sensor measurements of the physical process are monitored by decision logic that determines if a process violation has occurred, as illustrated by Figure 5.1. If a violation is detected, the decision logic switches control to a high-assurance and presumably slower version of the controller until the system is stabilized. A limitation of this scheme is that system stability cannot always be recovered as the controller fault is not detected until after it has caused the physical process to deviate from allowed operational limits.

Dai et al. advanced a fault detection architecture based on observing controller responses to new sensor inputs [6]. Physical process measurements are sent to both the regular high-performance version of the process controller and a trusted benchmark version of the controller algorithm. The responses of both controllers are used to generate a residual to determine if a controller fault has occurred, as shown in Figure 5.2. Unfortunately, the physical process is already affected by the erroneous controller output by the time the controller fault is detected and corrective actions, such as switching over to a high-assurance version of the controller, can occur. This may result in the inability to return the system to a stable state before damage is incurred.

Cárdenas et al. presented a physical model-based attack detection method with foundations in anomoly-based intrusion detection theory for computer systems and networks [3]. The specific threats addressed with this protection scheme include embedded controller intrusion

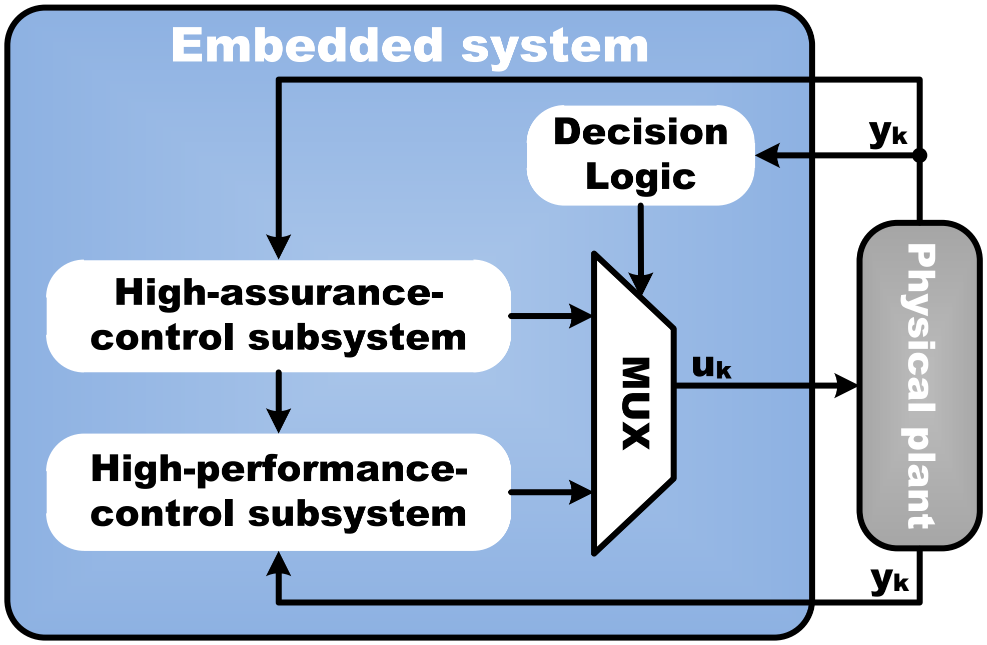

Embedded system
Decision Logic
High-assurance-control subsystem
High-performance-control subsystem
MUX
u_k
y_k
y_k
Physical plant

Figure 5.1: Plant fault detection [13]

attacks arising from compromised plant sensor data. Instead of using models of network traffic or software behavior, physical system models are used to develop a change detection-based intrusion detection algorithm. An embedded system implementation of a physical plant linear model runs concurrently with the plant, as illustrated in Figure 5.3. Controller responses are sent to both the physical plant and the embedded model. An anomaly detection module is then used to compare how sensor data measured from the physical plant compares to the response of the embedded plant model. When no differences are detected, the physical plant measurements are sent to the embedded controller which can then compute the feedback response. Sensor data intrusion is suspected when a difference is detected, in which case the embedded model's estimated physical plant state is sent to the embedded controller in an effort to filter out compromised measurements and continue plant control. This work develops a rigorous analysis and classification of PCS intrusion attacks and associated detection methods. However, it does not provide a means to detect and circumvent direct threats to controllers themselves such as configuration updates and Trojans.

Although many security solutions have been proposed for legacy embedded systems [2], these solutions are not optimized for process control applications. Design-time security techniques are very expensive and may not anticipate all system vulnerabilities. Such techniques

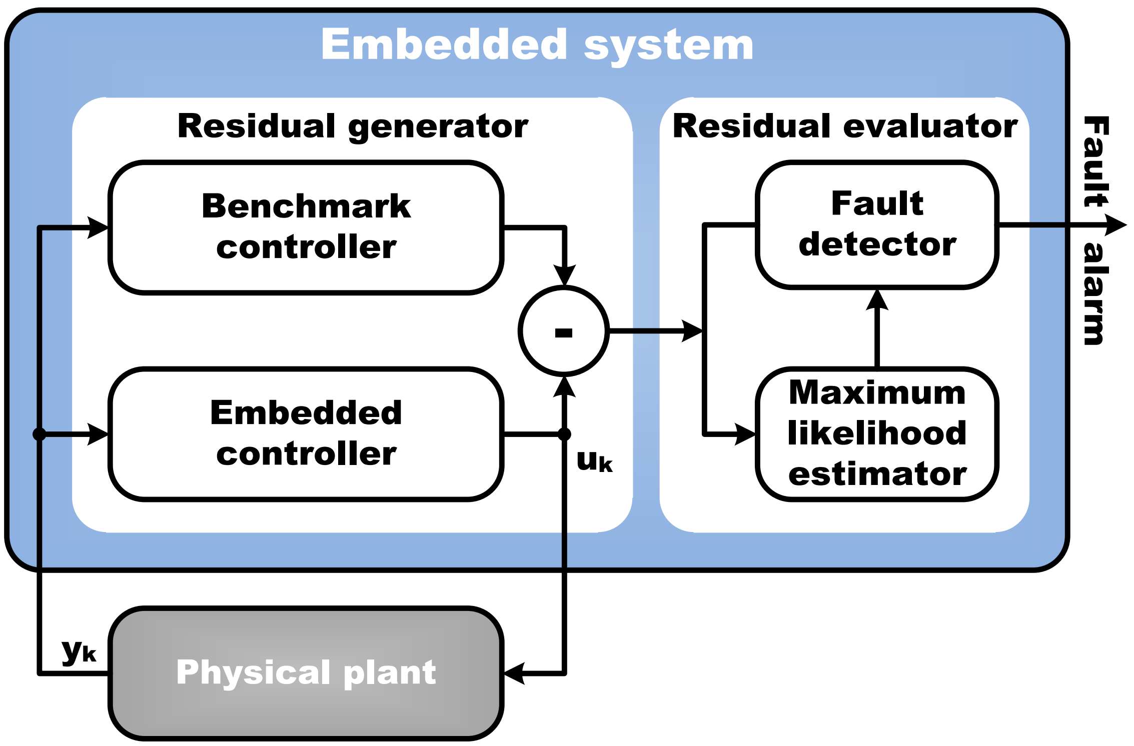


Figure 5.2: Controller fault detection [6]

often do not address vulnerabilities raised by software patches and updates, hardware reconfigurations, and zero-day exploits. This leads to a demonstrated possibility of controllers being surreptitiously compromised. An alternative approach is admitting the possibility of unanticipated threats and trying to cope with them using run-time security solutions. However, most existing run-time solutions are reactive, and can only detect erroneous controller behavior after its occurrence. Such detection methods may allow a physical processes to become unstable before corrective action can be taken. These techniques also tend to be threat-specific, leading to increased overheads for integrated solutions.

5.4 Controller Attack Prediction and Preemption

Our research stems from the observation that novel, deeply embedded protections are needed to cope with Stuxnet-class threats to process control systems. The specific goal of the work in this paper is to protect embedded controllers from configuration threats using a run-time security architecture synthesized with the DFSTAR methodology. Trust is neither required

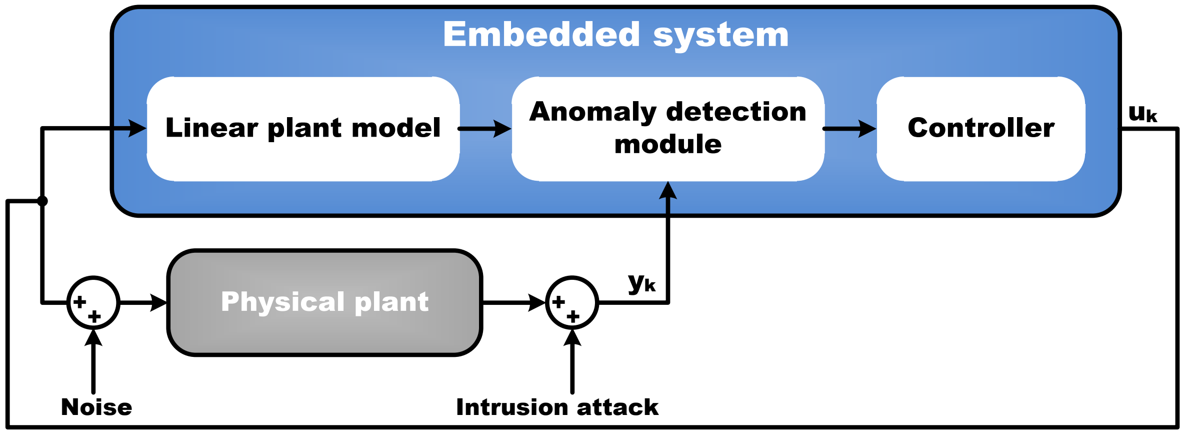


Figure 5.3: Controller intrusion attack detection [3]

in the active controller-under-protection nor its update and communication infrastructure, and applies only to a small set of simple, self-contained, synthesized, and verifiable CHARE add-ons. A CHARE root-of-trust is established to ensure an application's security and reliability specifications are being observed, and essentially serves the role of an ideal control room operator. Specification guards enable the root-of-trust to directly monitor system operation and override the controller-under-protection. The DFSTAR methodology incorporates security enhancements to the system structure and automatic tool extensions to the existing design flow.

Our threat model does not distinguish between hardware faults, software bugs, and malware such as Trojans since the common denominator is non-compliant controller behavior. A Stuxnet-like threat can hide itself using sophisticated means, but is less able to disguise its ultimate goal of disturbing system stability. Regardless of how the threat originates, the role of trusted protection system is to anticipate and deter consequences to the controlled process. Based on this philosophy, we present a novel method to predict and preempt erroneous behavior in physical process control. For the PCS domain, specifications for normal system behavior are already known, and accurate models for the controlled process usually exist. Our solution is complementary to other approaches that try to validate the design or prevent malware infiltration, and serves as a last line of cyber-defense against various threats to embedded controllers.

Physical systems and processes are characterized by quantitative temporal properties such as process response time, actuator delays, and sensor time constants. These physical latencies are not inherent in system models. The vast majority of physical processes can be described and modeled as linear time invariant systems with a high degree of accuracy under very realistic assumptions. Often, a plant model running on an embedded processor can be executed much faster than a real plant operating in a physical system. In a PCS, an embedded controller responds to variations in the state of a physical plant in order to maintain system stability. Execution speed of an embedded controller corresponds to the temporal characteristics of the associated physical process. The typical operating frequency of a digital embedded system controlling a physical process is proportional to the sampling rate of the physical process. Our approach to detect erroneous behavior of embedded controllers exploits potential speed differences between a physical plant and its model, which is analogous to the difference between running a physical system and simulating it.

The main idea of our approach is examining what the controller implementation will try to do in the future by embedding a second instance of the controller with an accelerated model of the plant. The model can be implemented in either hardware (such as an FPGA) or software (perhaps on a separate processor) depending on the required speed-up. The second controller instance can be identical to the original controller and implemented on the same platform. To maintain convergence with the physical system, the model's state is periodically synchronized with the plant's state. The embedded controller instance should be subject to the same conditions as the active controller by synchronizing the model's input with the system reference input, and applying the same patches and updates to both instances. A redundant embedded subsystem incorporating these measures can accurately predict the behavior of an embedded controller for several time steps in the future.

The operation of the added protection system is illustrated in Figure 5.4. During regular operation, the prediction unit continuously scans projected states of the active control algorithm against the corresponding projected states of the physical process. If a fault or Trojan activity causing the physical process to deviate from allowable bounds is detected by the specification guards, the root-of-trust immediately transfers control from the active controller to the high-assurance, stability-preserving controller, such as that used in [13]. The root-of-trust is also used to interface with all new configuration and parameter updates to the active controller. When a new update is received, it is first tested in the prediction unit using the current process state as a basis. The update is rejected if the prediction unit detects a deviation from allowable process bounds during initial screening. The update is only applied to the active controller when no violations are projected. After the update has been applied, the root-of-trust resumes predictive monitoring of the active controller to deter latent attacks introduced through the new update.

5.4.1 Prototype Controller Architecture

Some of the relevant, basic concepts of feedback control and modern control systems are presented in [7, 9]. In order to enhance the security, trust, and reliability of an embedded PCS, the existing system is augmented with a root-of-trust synthesized from a process model and specifications. As shown in Figure 5.5, major components of the predictive and preemptive architecture are:

- The original controller module containing an active controller-to-be-protected, a high-assurance, stability-preserving controller, and a mechanism to switch between them. This embedded system module runs at the typical sampling rate of the physical process.
- A prediction module consisting of a process model and a second instance of the active controller. This subsystem runs n times faster than the active control system module.
- A CHARE module that wraps the controller and prediction modules. This subsystem consists of specification guards as well as specialized model synchronization and timing blocks.

CHARE specification guards can be used to monitor either the physical process or the controller module input/output activity to assure compliance with the desired behavior of the physical process or a high-assurance benchmark controller. In this architecture specification guards monitor the process model output, as shown by Figure 5.5. Detection of anomalous behavior in the predictive subsystem triggers the guards to switch from the active controller of the physical process to a high-assurance controller. Recursion is possible with more than one backup controller and their predictive counterparts. The specialized synchronization unit is responsible for periodically updating the state of the model with the estimated state of the physical process. A specialized sample and hold unit updates the predictive subsystem input with the physical process reference input. The timing unit is responsible for clock generation and time emulation for the predictive subsystem.

The specification guard attached to the prediction module contains a maximum likelihood detector and a fault detector to predict faults before they actually occur in the controller-

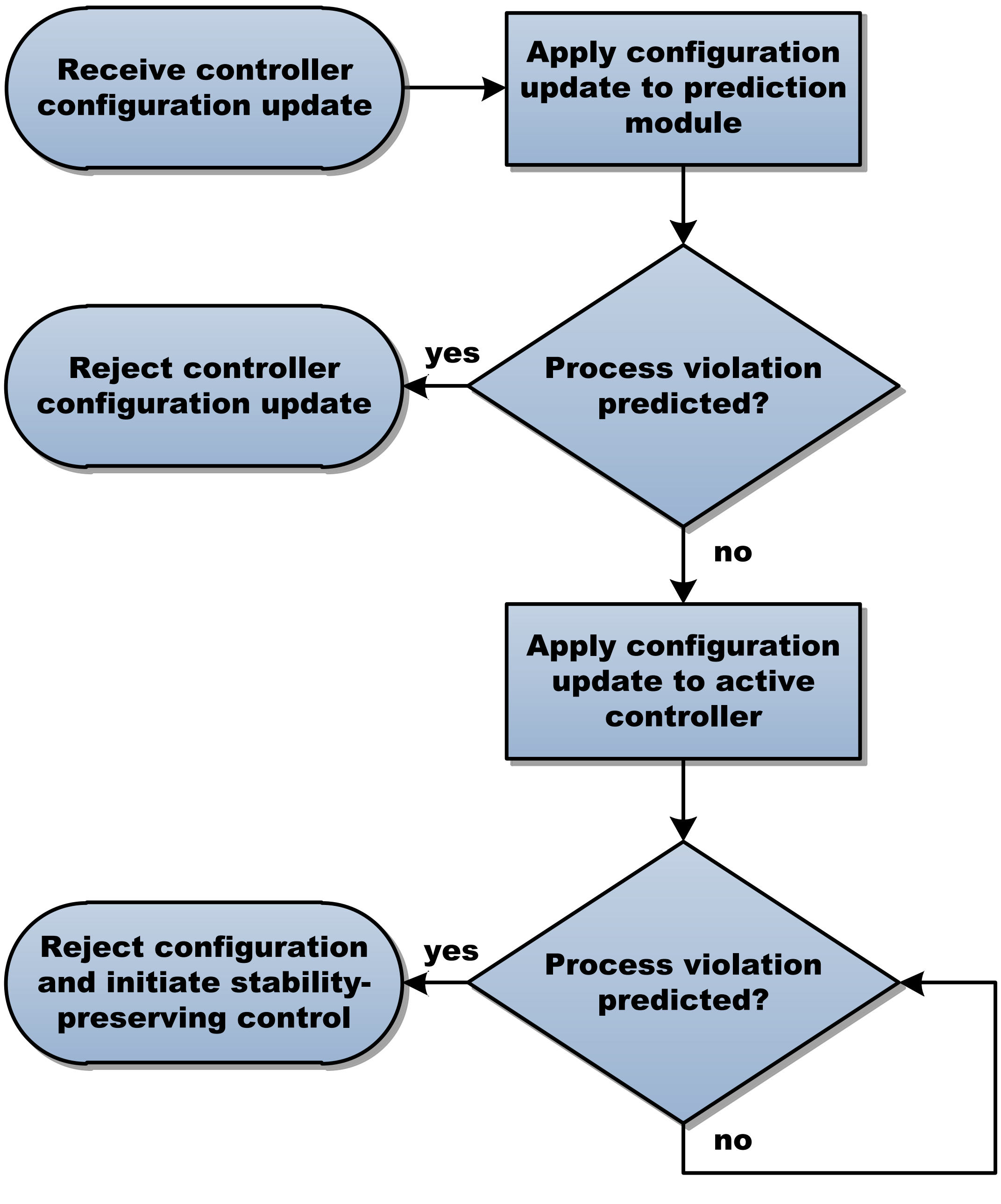

Receive controller configuration update
Apply configuration update to prediction module
Process violation predicted?
yes
Reject controller configuration update
no
Apply configuration update to active controller
Process violation predicted?
yes
Reject configuration and initiate stability-preserving control
no

Figure 5.4: Procedure for testing controller configurations

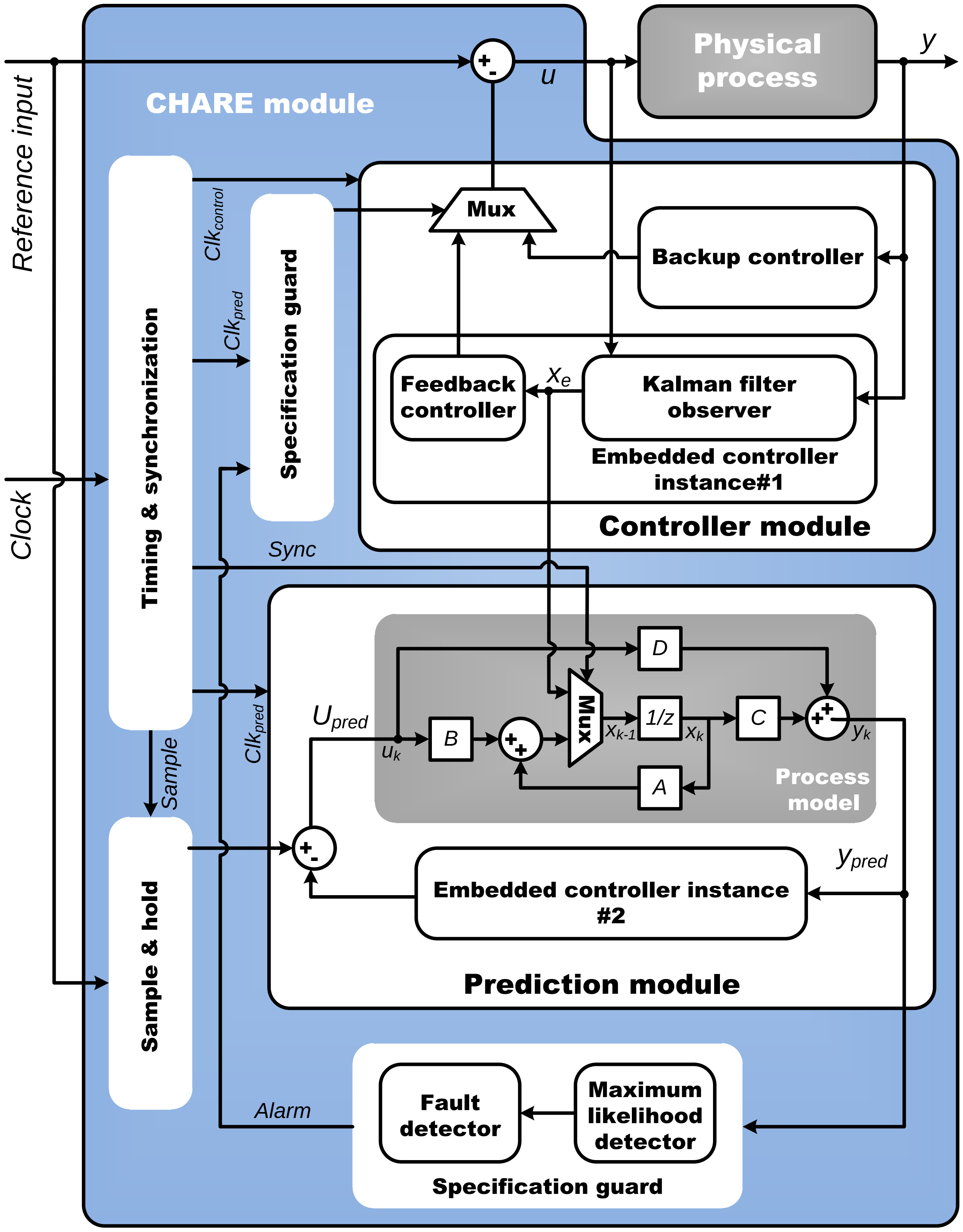


Figure 5.5: Predictive and preemptive security architecture

under-protection. Theoretically, if the predictive controller is secure and no threats are affecting it, the output of the controller will conform to the normal operating criteria described by the security policies embedded in the fault detector module. Any threat affecting the prediction module controller will show up later in the actual controller and, if not preempted, will increasingly affect the predictive controller's output. In practice, many factors other than those related to security threats can cause such a deviation from the normal operating conditions, such as mistuning of the controller parameters and the random noise resulting from the process. To address the challenge of distinguishing faults resulting from cyber-attacks from noise, accurate descriptions of the controller characteristics, operating conditions, and the statical distribution of noise are needed.

The fault detector does not rely on a single sample to decide the controller's integrity. For PCSes, it has been shown that as the number of the observed samples increases, the statistical distribution of the process noise becomes Gaussian with constant mean and variance values [6]. Consequently, the statistical distribution of the controller's output follows the Gaussian noise distribution as it is the only random variable in the output equation. Deviation from the normal operating conditions caused by either cyber-threats or controller faults shifts the predictive controller's output mean and variance computed over a significant number of samples to new values outside the range defined by the security policy. In other words, deviation of the output root mean square (RMS) value from nominal bounds indicates a fault or attack in the controller-under-protection. The maximum likelihood detector unit shown in Figure 5.5 is used to evaluate the RMS value of the embedded model's output in the prediction module. The fault detector then tests the predictive controller's output against the the RMS value generated by the maximum likelihood detector to determine if the likelihood ratio lies within the predefined threshold range captured by the security policies.

5.4.2 Timing and Synchronization

Our approach advances new terminology such as: the time scaling factor n which indicates the predictive subsystem speed-up; the prediction window *W_pred* which denotes the foreseen time period; and synchronization time *T_sync* which determines the updating frequency of the model's state in the predictive subsystem. *W_pred* is function of n and *T_sync*, as shown by equation 5.1. *T_sync* is application-dependent whereas n is both application- and platform-dependent. Assuming flexibility in assigning n and *T_sync*, increasing n improves

W_pred at zero cost in terms of the updating frequency, while increasing T_sync augments W_pred on the expense of reducing the updating frequency. T_sync is often the more flexible and tunable parameter when significant changes to W_pred are needed. Multiple trade-offs must be evaluated when assigning values of n and T_sync where the physical process characteristics and the embedded platform features are the assignment criteria.

$$W\_pred = n \cdot T\_sync \tag{5.1}$$

Time scaling is accomplished by applying modifications to both system and input signals. System modifications vary for continuous- and discrete-time models of the physical process. For a continuous-time model, time scaling is achieved by multiplying system state space matrices by the desired time scaling factor n. For discrete-time embedded systems employing digital controllers, scaling down the sampling time of the physical process by a factor of n automatically scales down the time of the system internal signals. Input signal time cannot be scaled down because this requires prior knowledge of signal contents. To tackle this problem, the model's input can be periodically synchronized with the reference input at the physical system sampling rate by assigning T_sync to be $T\_sampling$ seconds. However, this approach limits the prediction window to $n \cdot T\_sampling$ seconds.

Another approach can be adopted where the process model and the physical system are synchronized whenever the reference input to the physical system is changed. Such an approach produces an adaptive prediction window, which may not be preferred for security reasons. In a PCS, the reference input to a physical process is often the desired stable output of the system which implies that reference input changes are limited in terms of both amplitude and frequency. This implies that a sample and hold technique can be used to periodically update the model's input with the reference input without the need for an adaptive synchronization method. We adopt this approach to establish a security scheme with a controllable synchronization time and a fixed prediction window.

We consider both event-driven and time-driven faults. Accurate detection of event-driven faults depends on the proper and frequent updating of the model's state in the predictive subsystem. Figure 5.5 shows the switching technique created to update the model's state x_k with the estimated plant's state x_e generated by the Kalman filter state observer. The model's state updating frequency is a function of the desired prediction window and the model's input synchronization scheme. Predictive subsystem time must emulate the real time in order to successfully detect time-driven faults and properly operate time-driven modules and processes. Time emulation requires generating the predictive subsystem time in terms of n and T_sync, and relating it to the real time t. The predictive subsystem time is directly proportional to n, whereas T_sync formulates the reference time base which periodically resets the predictive time T_pred to the real time t as shown by equation (5.2). Figure 5.6 illustrates the predictive subsystem time for the case study described in Section 5.5 where the the time scaling factor n is 10, and two values of T_sync (1 and 10 seconds) are used for different prediction windows.

$$T\_pred = T\_sync \cdot \lfloor \frac{t}{T\_sync} \rfloor + n \cdot \text{mod}(\frac{t}{T\_sync}) \tag{5.2}$$

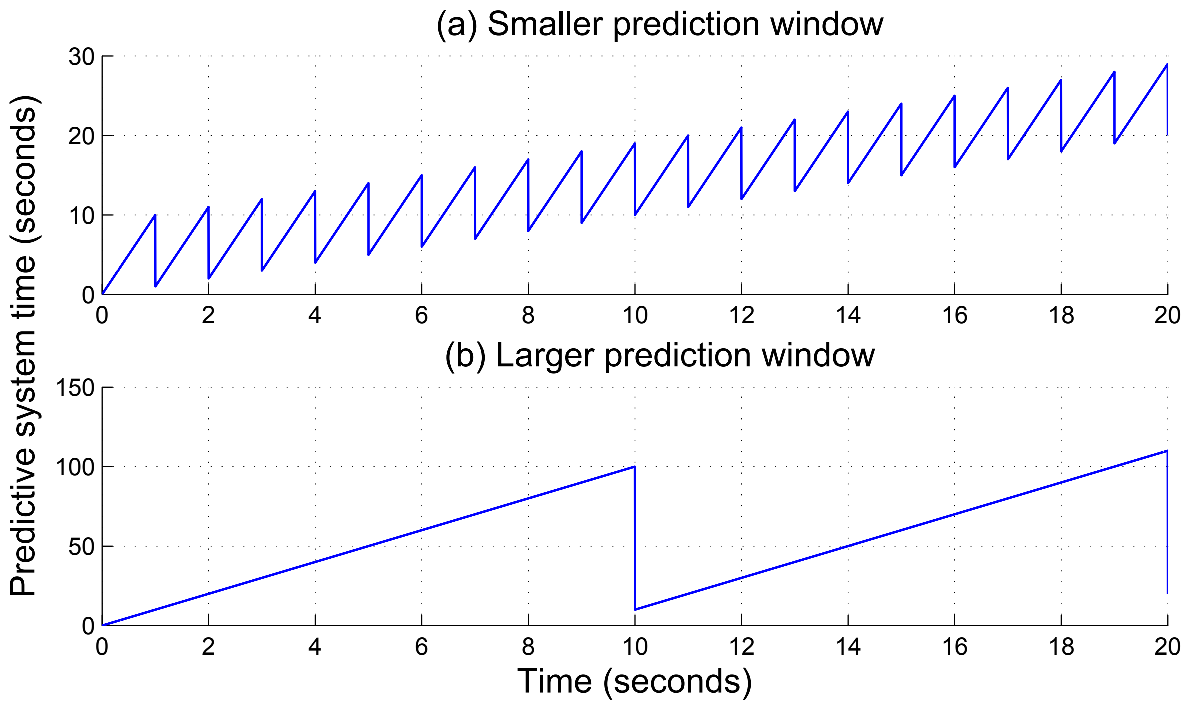


Figure 5.6: Relationship between real time and predictive subsystem time

5.4.3 Model-based Design Flow Enhancements

Figure 5.7 illustrates a prototype design and implementation flow to create a root-of-trust for an embedded controller-under-protection. We focus on model-based design used to generate solutions for problem domains that have a well-established mathematical basis, such as process control, signal processing, and communications. Although model creation and analysis are a routine part of engineering, models are mostly used to explore and validate abstract solutions such as structures and algorithms, while the actual solution is implemented from scratch. In contrast, model-based design automatically synthesizes the solution—usually either hardware or software—directly from the model.

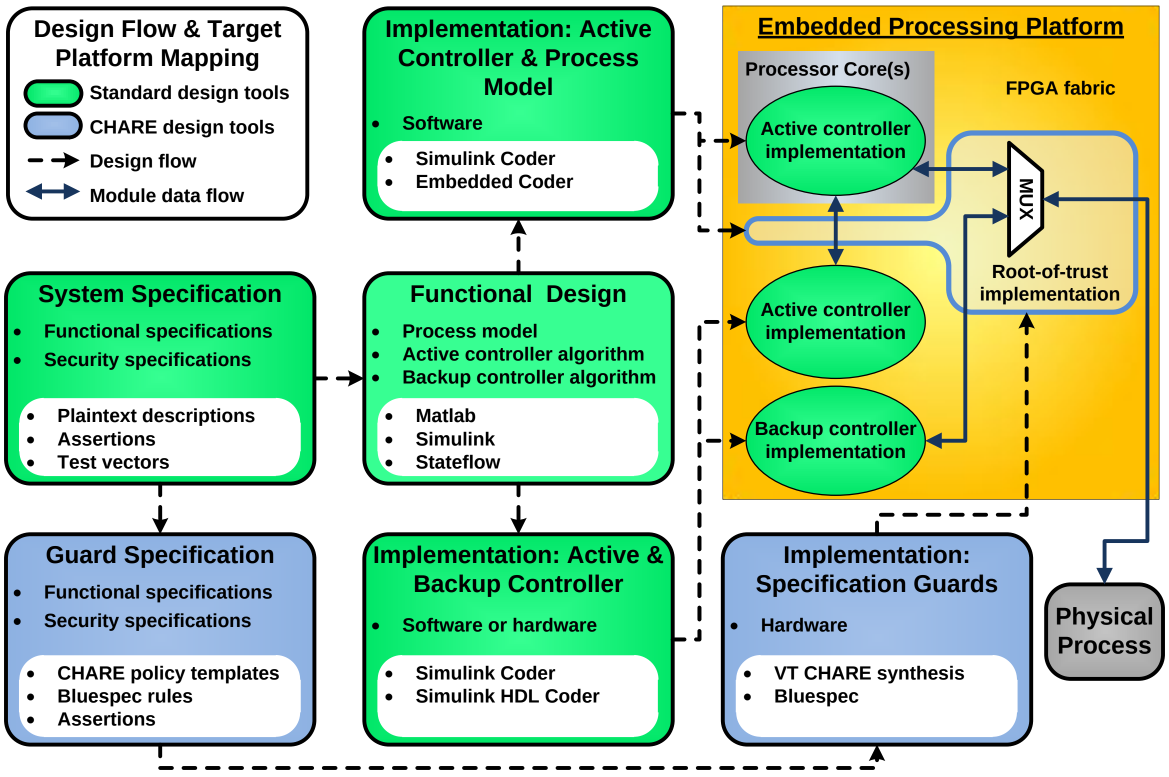


Figure 5.7: CHARE design flow

Controller design begins with functional specification, often initially captured as plaintext documentation or assertions of expected behaviors of the system for the the intended application. Specifications reflecting application-specific security or reliability policies for the controller and physical process can similarly be developed at this time. Specifications guide the functional design of control algorithms in a tool such as MATLAB Simulink or Stateflow. The process model's structure is often captured graphically also using these tools in an effort to evaluate control algorithms. Tools such as Simulink Coder or HDL Coder then automate software and hardware generation of the controller, which is mapped to embedded platform components such as dedicated processors or configurable FPGA fabric.

The root-of-trust in Figure 5.7 consists of the process control violation prediction and

CHARE modules detailed in Figure 5.5. The model developed to describe the process can be reused in the prediction module and implemented with the same tools used to synthesize the controller. An optimized implementation of the process model helps to reduce the time and space overheads of our predictive subsystem, which is especially important in embedded control environments that do not use PC-class hosts. As with specifications, this design flow assumes that the process model is accurate and can be trusted. Fortunately, process and high-assurance backup controller models tend to be stable, synthesized, self-contained, and subject to formal verification.

Functional and security policy specifications are inputs to the design flow for creating CHARE specification guards, as shown in Figure 5.7. Application-independent specification languages are used to prepare policies for synthesis, such as acceptable ranges for process sensor and controller outputs. Clearly specifying permitted ranges and rates-of-change for process and controller I/O specifically guards against Stuxnet-like attacks on processes requiring smooth control changes.

Though the DFSTAR methodology is neither hardware- nor software-specific, we choose to focus primarily on the development of root-of-trust hardware. A hardware-oriented solution provides the access and performance necessary to implement run-time protections with increased tamper resistance [11]. An example of CHARE implemented on a modern embedded processing platform marketed for industrial control applications is shown in Figure 5.8 [16]. The CHARE root-of-trust is synthesized to programmable hardware fabric. Processes to be monitored, such as the active controller and prediction module, can be hosted on the embedded processor cores.

The specification guard components to be validated are simple, independent, and largely synthesized from high-level abstractions, such as SystemVerilog assertions or Bluespec SystemVerilog rules with guarded atomic actions, as described in [8]. Our future work will explore methods for generating relevant assertion automata guards in detail. For the sake of verification, Simulink wrappers can be generated to enable simulation of the synthesized predictive subsystem, synchronization, and switchover blocks.

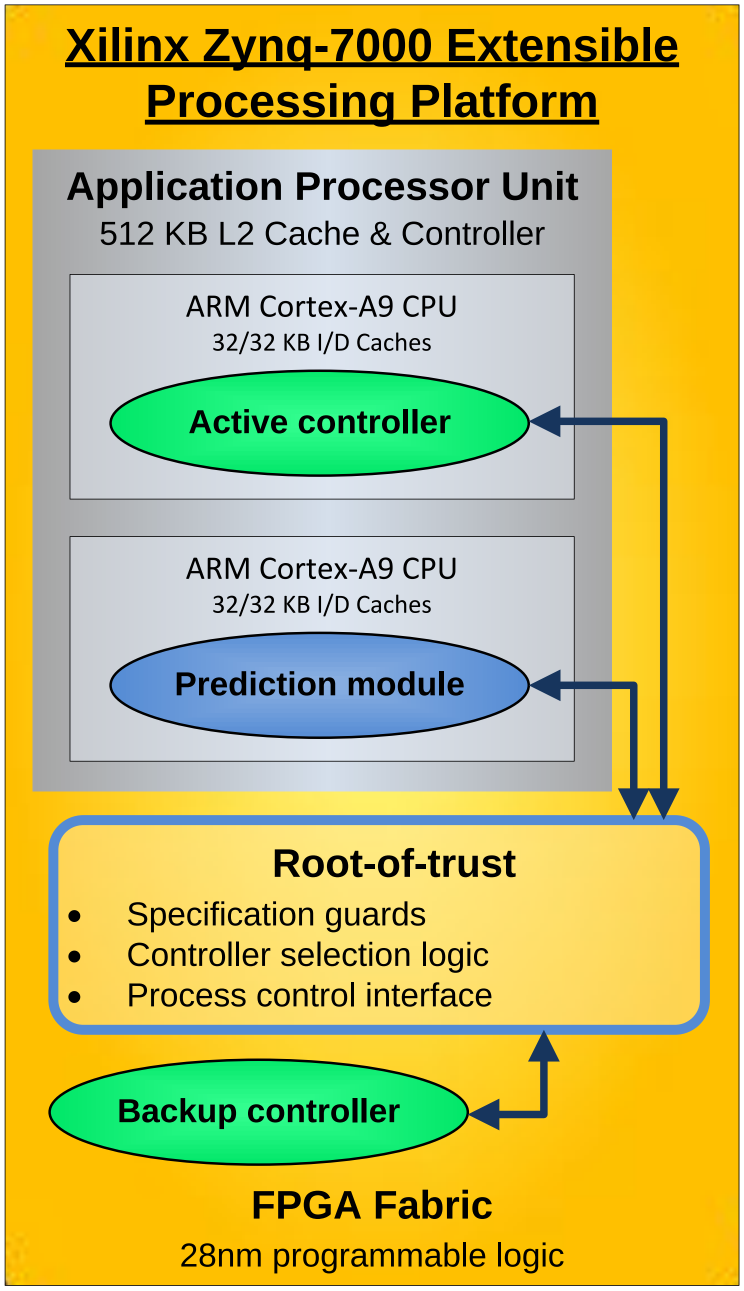


Figure 5.8: CHARE implemented on a Xilinx Zynq-7000 Extensible Processing Platform

5.5 Example Control Application

In order to illustrate and evaluate our approach, an aircraft autopilot pitch controller is used as a case study [12]. Flight control is a safety-critical application where controller faults can have catastrophic consequences. Linear Quadratic Gaussian (LQG) control is a modern approach adopting time-domain analysis, state space representations, and state observers to enhance the control process. It concerns uncertain linear systems disturbed by additive white Gaussian noise and undergoing control subject to quadratic costs [14]. LQG controllers are widely deployed, and their structure helps to present our concepts and architecture effectively. Nevertheless, our approach is still applicable to other control techniques.

5.5.1 Pitch Control Process Model

The motion of an aircraft is governed by a set of six non-linear differential equations. These equations can be decoupled into longitudinal and lateral equations under certain assumptions [12]. The pitch angle is a third-order longitudinal problem and is controlled by adjusting the angle of the rear elevator. Figure 5.9 shows the basic coordinate axes and forces acting on an aircraft.

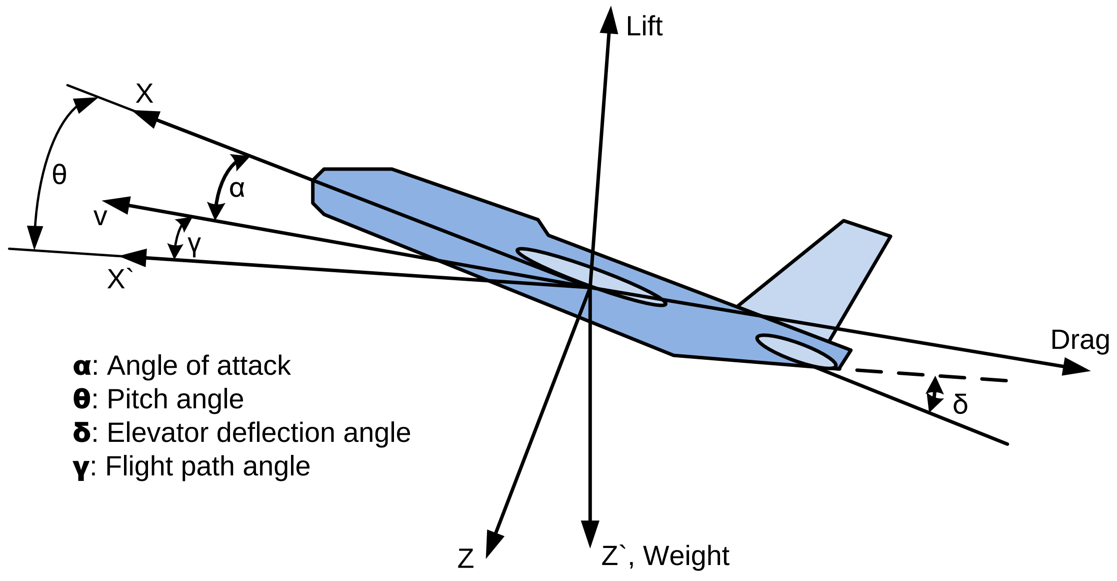


Figure 5.9: Coordinate axes and forces acting on an aircraft [12]

As described in [12], the equations of motion of a Boeing commercial aircraft can be written as:

$$\begin{aligned}\dot{\alpha} &= -0.313\alpha + 56.7q + 0.232\delta_e \\ \dot{q} &= -0.0139\alpha - 0.426q + 0.0203\delta_e \\ \dot{\theta} &= 56.7q\end{aligned}$$

where α is the angle of attack; q is the pitch rate; q is the pitch angle; and δ_e is the elevator deflection angle.

Using the differential equations controlling the plane motion, the state space representation of the pitch angle system is as follows:

$$\begin{bmatrix}\dot{\alpha} \\ \dot{q} \\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}-0.313 & 56.7 & 0 \\ -0.0139 & -0.426 & 0 \\ 0 & 56.7 & 0\end{bmatrix}\begin{bmatrix}\alpha \\ q \\ \theta\end{bmatrix} + \begin{bmatrix}0.232 \\ 0.0203 \\ 0\end{bmatrix}\begin{bmatrix}\delta_e\end{bmatrix}$$

$$y = \begin{bmatrix}0 & 0 & 1\end{bmatrix}\begin{bmatrix}\alpha \\ q \\ \theta\end{bmatrix} + \begin{bmatrix}0\end{bmatrix}\begin{bmatrix}\delta_e\end{bmatrix}$$

In the presence of noise, this equation can be expressed in a state space form:

$$\begin{aligned}\dot{x} &= Ax + Bu + \omega_{proc} \\ y &= Cx + Du + v_{sensor}\end{aligned}$$

where x is a column matrix composed of α, q, and q elements representing system's state; the input u is the elevator deflection

5.6 Conclusions and Current Work

Comprehensive rather than point solutions are needed to help PCS infrastructure withstand an emerging malware onslaught. As illustrated by Stuxnet, preventing malware infiltration is difficult in complex, networked control systems having zero-day exploits. Trojans may also arise from the global supply chain and use of third-party IP. This leads to a demonstrated possibility of controllers being surreptitiously compromised. Erroneous controller behavior must be detected before it critically affects a physical process.

Existing solutions to run-time bug and fault detection include monitoring the process state arising from past controller actions, or comparing present outputs from independent controllers. Our run-time system includes a second instance of the active controller connected to a model of the plant giving a short-term projection of future controller actions and process state. The model's state is periodically synchronized with the plant's state to prevent divergence. Erroneous controller behavior is detected before it affects the physical process, allowing preemptive alarms or actions.

The blocks conferred with trust should be minimal in complexity and number so that synthesis and formal verification methods may be applied. In addition, these blocks should have rigorous update procedures, and use distinct software and hardware resources. Ideally the trusted blocks do more than just protect against malware by also enhancing reliability in the presence of software bugs and hardware faults. The DFSTAR methodology meets these goals by exploiting: (1) the inherent controllability and observability of a PCS; (2) the existence of specifications for normal PCS operation; and (3) the model-based design approach commonly used during PCS development.

Tools are under development to automatically generate the trusted components described in this paper. FPGA platforms enable the use of integrated yet independent resources for monitoring functions, and allow both hardware and software implementation of synthesized blocks. Once the tools are complete, we will be able to assess the design-time and run-time overheads of the DFSTAR methodology. After targeting simple controller applications such as pitch control, we will look at inserting a system monitor in a network of process controllers. We also plan to see how these tools scale up to to more complex multiple-input-multiple-output (MIMO) controllers used in modern PCS platforms.

Acknowledgments

The CHARE framework used in this work was developed in the AUSTIN (An Initiative to Assure Software Radios have Trusted Interactions) project supported by the National Science Foundation, Grant No. CNS-0910557.

References

[1] M. Brundle and M. Naedele. Security for process control systems: An overview. *Security Privacy, IEEE*, 6(6):24–29, Nov–Dec 2008.

[2] A.A. Cárdenas, S. Amin, and S. Sastry. Secure control: Towards survivable cyber-physical systems. In *Distributed Computing Systems Workshops, 2008. ICDCS '08. 28th International Conference on*, pages 495–500, Jun 2008.

[3] Alvaro A. Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. Attacks against process control systems: risk assessment, detection, and response. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'11, pages 355–366, 2011.

[4] T.M. Chen. Stuxnet, the real start of cyber warfare? [editor's note]. *Network, IEEE*, 24(6):2–3, Dec 2010.

[5] F. Cohen. Automated control system security. *Security Privacy, IEEE*, 8(5):62–63, Sep–Oct 2010.

[6] C. Dai, S.H. Yang, and Liansheng Tan. An approach for controller fault detection. In *Fifth World Conference on Intelligent Control and Automation (WCICA)*, volume 2, pages 1637–1641, Jun 2004.

[7] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 9th edition, 2000.

[8] Mohammed M. Farag, Lee W. Lerner, and Cameron D. Patterson. Thwarting software attacks on data-intensive platforms with configurable hardware-assisted application rule enforcement. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 207–212, Sep 2011.

[9] Gene F. Franklin, David J. Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001.

[10] Steven L. Kinney. *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology).* Newnes, 2006.

[11] Lee W. Lerner, Mohammed M. Farag, and Cameron D. Patterson. Interacting with hardware Trojans over a network. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, June 2012.

[12] W.C. Messner and D.M. Tilbury. *Control tutorials for MATLAB and Simulink: User's Guide.* Addison-Wesley, 1998.

[13] Lui Sha. Using simplicity to control complexity. *Software, IEEE*, 18(4):20–28, Jul–Aug 2001.

[14] L.M. Surhone, M.T. Tennoe, and S.F. Henssonow. *Optimal Projection Equations.* VDM Verlag Dr. Mueller AG & Co. Kg, 2010.

[15] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *Design Test of Computers, IEEE*, 27(1):10–25, Jan–Feb 2010.

[16] Xilinx, Inc. *Zynq-7000 EPP Overview, DS190 (v1.1.1)*, June 2012.

Chapter 6

Thwarting Software Attacks on Data-intensive Platforms with Configurable Hardware-assisted Application Rule Enforcement

6.1 Abstract

Security is difficult to achieve on general-purpose computing platforms due to their complexity, excess functionality, and resource sharing. An alternative is the creation of a Tailored Trustworthy Space for the system or application class of interest. We focus on data-intensive computing systems using reconfigurable hardware to implement streaming operations, and provide security assurances that are independent of application software, middleware, or operating system integrity and correctness. All interaction between software and the dataflow hardware passes through an automatically synthesized and formally verified hardware controller incorporating enforcement and real-time monitoring of application-specific rules. Abstractions provided by the Bluespec high-level language assist in the translation of domain-specific policy rules to synthesized logic. For the cognitive radio example used, hardware-enforced policies include physical layer rules such as sanctioned spectrum usage. Policy changes cause the secure generation and transfer of a new controller-wrapped datapath hardware plug-in. Datapath dynamic block swaps and cryptographic operations are

managed entirely by the hardware controller rather than software drivers. Design for performance and design for security are therefore simultaneously addressed since the datapath is configured and monitored at hardware speeds, and software has no access to datapath configurations and cryptographic keys.

6.2 Introduction

There is a trickle-down of architectural advances from high performance desktop and server computing platforms to embedded computer systems. Larger address spaces, more effective caches, RISC instruction sets, multi-gigabit serial I/O, multi-core and heterogeneous architectures are commonly used to satisfy performance and power constraints on systems performing computations on streaming data. However, embedded system adoption of desktop platform security mechanisms tends to lag adoption of performance techniques. Several reasons are usually given: power and cost concerns, a belief that software-based attacks are less likely compared to desktop platforms, and a focus on physical attacks. As embedded platforms are increasingly networked, the primary threat will likely shift to malware.

The trusted computing technologies used in large-scale, personal, and later embedded systems have the following progression:

1. *SLS: Software limits access to software and data.* Software is structured in layers, and application or user requests requiring services from a more privileged layer are vetted by software-implemented processes. The goal is to separate layers with robust APIs that cannot be circumvented. Java virtual machines and packet filters are examples of software stratification. Attacks generally have executable code masquerade as data, and use vulnerabilities such as buffer overflows to execute the data.

2. *SLH: Software limits raw hardware resource access.* Innermost software layers have exclusive responsibility for allocating and managing hardware resources such as CPUs, memory, and peripherals. Examples of layers interacting directly with hardware are virtual machines and operating systems. Access to hardware does not necessarily imply access to (possibly encrypted) software and data. Direct programmatic access to hardware is different from physical access to hardware, which introduces the possibility of side-channel and probing attacks.

3. *HLS: Hardware limits access to software and data.* Static hardware units, possibly controlled by the innermost and trusted software layer, assist in the separation of software processes and layers. Examples are memory management units (MMUs), Trusted Platform Modules (TPMs) [23], protected execution and launch portions of Intel's Trusted Execution Technology (TXT) [13], and the use of eTokens. Although hardware provides enforcement, trust in supervisory software may still be needed.

4. *HLH: Hardware limits raw hardware resource access.* Static hardware controllers have exclusive responsibility for managing hardware-implemented processes or channels, and can deny requests from software at any layer. Examples are the protected input and graphics parts of Intel's TXT, Intel's Virtualization Technology [14], and hardware firewalls.

The programmability and performance of reconfigurable hardware suits data-intensive embedded computing applications. As shown in Fig. 6.1, traditional computing platforms have a fixed trust hierarchy consisting of static hardware, operating system, middleware, and application software. However, trust inheritance becomes complicated in configurable platforms when software updates the underlying hardware structure. "In hardware we trust" is no longer axiomatic since the hardware can be modified to violate specific policies. Current practice places dynamic hardware configuration under the control of application-level software, and even proposed OS-managed reconfiguration remains an SLH solution in the above taxonomy. Dynamic hardware blocks are application-tailored and potentially untrusted. Software modification of hardware structure is analogous to self-surgery, and independent hardware should provide oversight rather than rely solely on the correctness and integrity of application software and circuits. We insert a hardware-implemented, application-specific controller and monitor on the boundary between static hardware (which hosts software) and dynamic application hardware. This HLH approach retains most of the flexibility of application software directly controlling dynamic hardware, while enhancing trustworthiness, performance, and hardware abstraction.

The remainder of the paper is structured as follows: Section 6.3 surveys existing and proposed approaches to enhancing platform security. Section 6.4 describes an architecture combining a reconfigurable datapath with a synthesized controller to enforce and monitor application-specific rules. The use of the platform and associated tools are illustrated in Section 6.5 with a cognitive radio application. Finally, Section 6.6 draws conclusions.

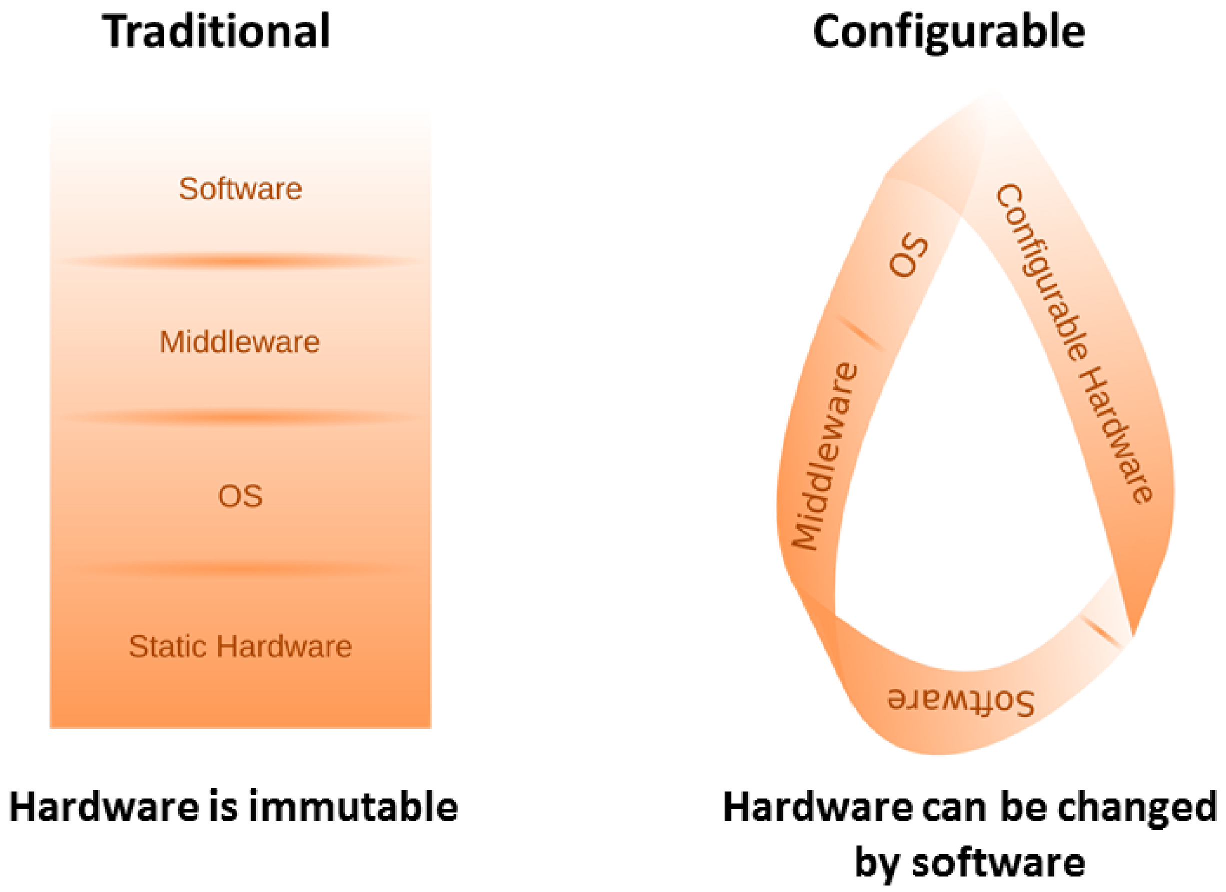


Figure 6.1: Static versus reconfigurable platforms

6.3 Existing Security Approaches

Current support for secure computing uses fixed hardware to enforce resource separation between applications. The Trusted Computing Group (TCG) developed a trusted platform (TP) specification to provide consistent behavior for a specific purpose using software and hardware enforcement [23]. Root of Trust for Management (RTM), Trusted Platform Module (TPM), and Trusted Software Stack (TSS) components provide three basic features: protected capabilities, attestation, and integrity measurement and reporting. Microsoft's Next Generation Secure Computing Base (NGSCB) is a software architecture exploiting the security provided by a TPM. The NGSCB consists of two kernels: an untrusted mode kernel, and a trusted NEXUS mode kernel that provides a secure environment for trusted code [19]. Intel's TXT uses processor enhancements, a TPM, operating system extensions such as the

NGSCB, and enabled applications to protect sensitive information from software-based attacks [13]. ARM's TrustZone processor extensions support normal and secure environments, with a monitor mode providing robust context switches [3]. In all of these commercial examples, the sole focus is software separation, the execution model does not consider hardware adaptability, and there is excessive reliance on software correctness and integrity [21].

Reconfigurable hardware has been used to implement policy-driven memory protection mechanisms [12]. This work develops an access policy language to describe fine-grained memory separation of modules on an FPGA. A policy compiler converts the specified memory access policies into enforcement hardware modules. Reconfigurable hardware has been used to implement a TPM [6]. Modifications to existing FPGA architecture are required, including updates to the AES core bitstream decryptor and adding an on-chip non-volatile memory. An FPGA has been augmented with a Trust Block consisting of a TPM, a secure ROM storing FPGA configuration data, and switch logic used to configure the FPGA solely from secure ROM during system boot [8]. Unfortunately, these efforts do not target platforms where the hardware structure potentially changes during operation.

Reconfigurable systems often use third party IP cores. Although ideally these cores would be verified by a trusted party, cost and source code requirements can make such a development model impractical. Reliance on off-the-shelf IP modules provided by multiple vendors with different levels of trust introduces serious security concerns [11]. A moat and drawbridge model provides spatial module isolation and statically verifiable communication flow [10], but does not address modules with self-contained trojan horses.

6.4 Minimizing Software and IP Trust in a Reconfigurable Platform

Separation is a fundamental tool in secure system design and should be used in reconfigurable platforms with hardware and software interactions. In such platforms, software control of hardware configuration introduces new security concerns compared to static hardware systems. System trust can be enhanced by enforcing application-specific access control policies using either software or hardware. Hardware, which has greater tamper resistance and is better suited to formal analysis than software, provides policy oversight in the Configurable Hardware-assisted Application Rule Enforcement (CHARE) platform.

As shown in Fig. 6.2, the CHARE architecture has four major components: an embedded application processor, a reconfigurable controller-wrapped datapath, dedicated hardware to securely configure the datapath, and secure access to a shared configuration server. Fig. 6.3 shows the separate regions allocated to these components on the Xilinx Virtex-5 FX130T FPGA used in the initial prototype, with controlled communication between regions. Xilinx's Embedded Development Kit (EDK) connects one of the embedded PowerPC 440 processors to peripherals over a Processor Local Bus (PLB). The processor runs real-time Linux for data-intensive applications implemented with both software and custom hardware. A general-purpose I/O (GPIO) control interface stores datapath update request parameters while buffers transfer data between software and the reconfigurable hardware. There are few static routes crossing the dynamic region since most of the I/O signals connected to the two processors reside in the leftmost I/O banks. Roughly 70% of the chip area, including all 320 DSP slices and the majority of the Block RAM, is allocated to the reconfigurable plug-in region.

The reconfigurable hardware block consists of a datapath wrapped in a Datapath Rule Enforcement Controller (DREC). Parameterized IP cores are connected to implement streaming algorithms in domains such as DSP, communication, and video. The DREC is a hardware-based finite state machine responsible for checking that software-issued datapath update requests conform to policy rules embedded in the DREC. Software-visible datapath update registers contained in the GPIO bus interface are not directly connected to the datapath, and may not even have a one-to-one correspondence with actual datapath parameters. Invalid update requests return an error, while requests conforming to policy rules can result in a parameter update, individual module swaps, or a complete datapath plug-in replacement.

The DREC also serves as a datapath hardware abstraction layer. This has two advantages: software interaction with the hardware is simplified to enhance portability, and the datapath's detailed implementation is not revealed to software. For example, software is oblivious to the possible use of reconfiguration for swapping cores in response to datapath update requests. The hardware model presented to software restricts the set of control capabilities to that of a virtual ASIC implementing only the configuration options currently authorized. An additional advantage of custom hardware plug-ins is the provision of software-independent cryptographic services. For example, data may be automatically encrypted or decrypted by the plug-in using a key embedded in the datapath controller. Configurable hardware is generally more efficient than software for cryptographic algorithms [18], yet can be changed

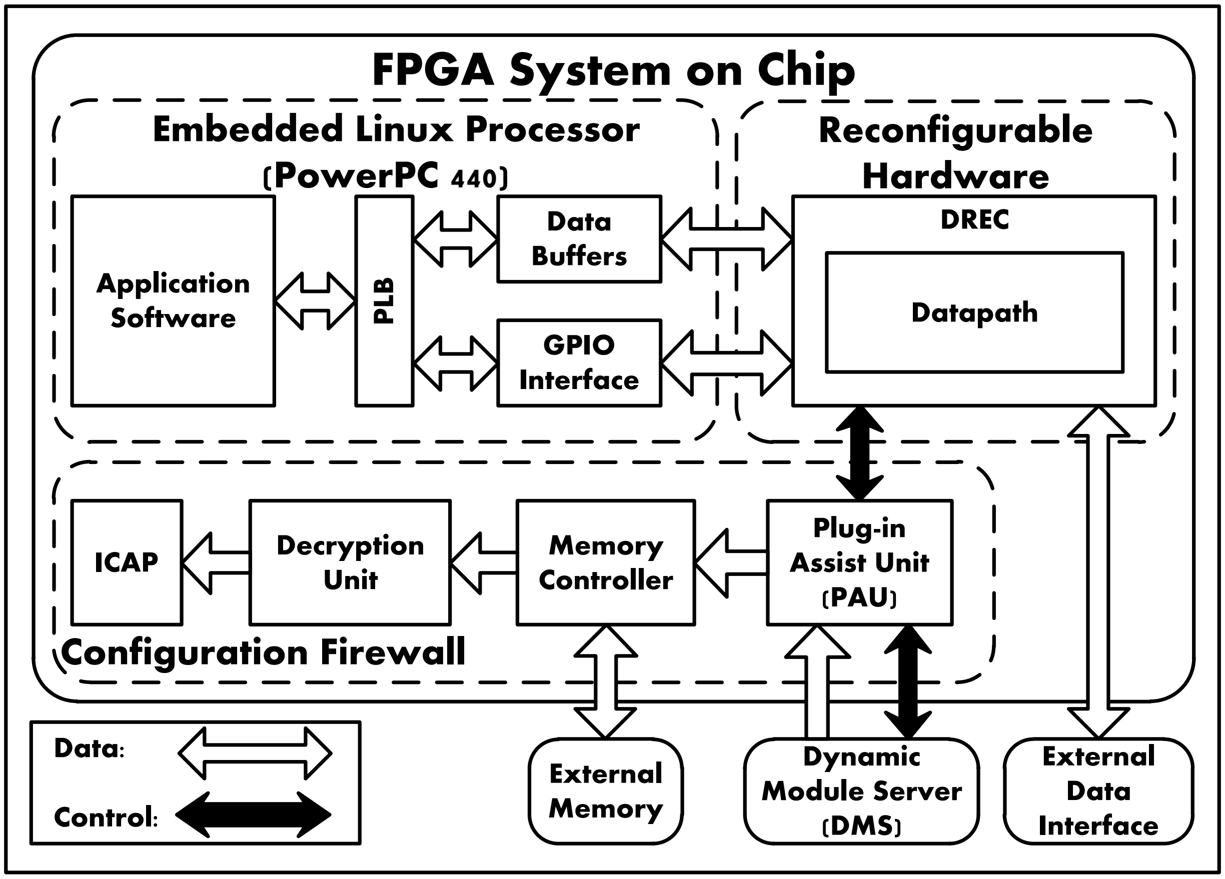


Figure 6.2: CHARE prototype platform

as readily as software implementations. Session keys may be used as a means of imposing expirations on particular capabilities.

DREC logic is never updated independently of the datapath, and any policy updates necessitate a complete plug-in replacement. Plug-ins may include monitors to check the operation of individual cores that may be untrusted or subject to single-event upsets. Detection of anomalous behavior signals the DREC to reload the plug-in if an upset occurred or a trojan horse was enabled. New plug-ins are securely (and perhaps wirelessly) transferred from a remote, shared and trusted Dynamic Module Server (DMS). Server-class hardware suits the time- and memory-intensive EDA tools required to generate new FPGA configurations; these tools exceed the resources available in embedded platforms. The DMS runs the PATIS tools to accelerate hardware plug-in implementation through the automatic parallel application of standard implementation tools [5].

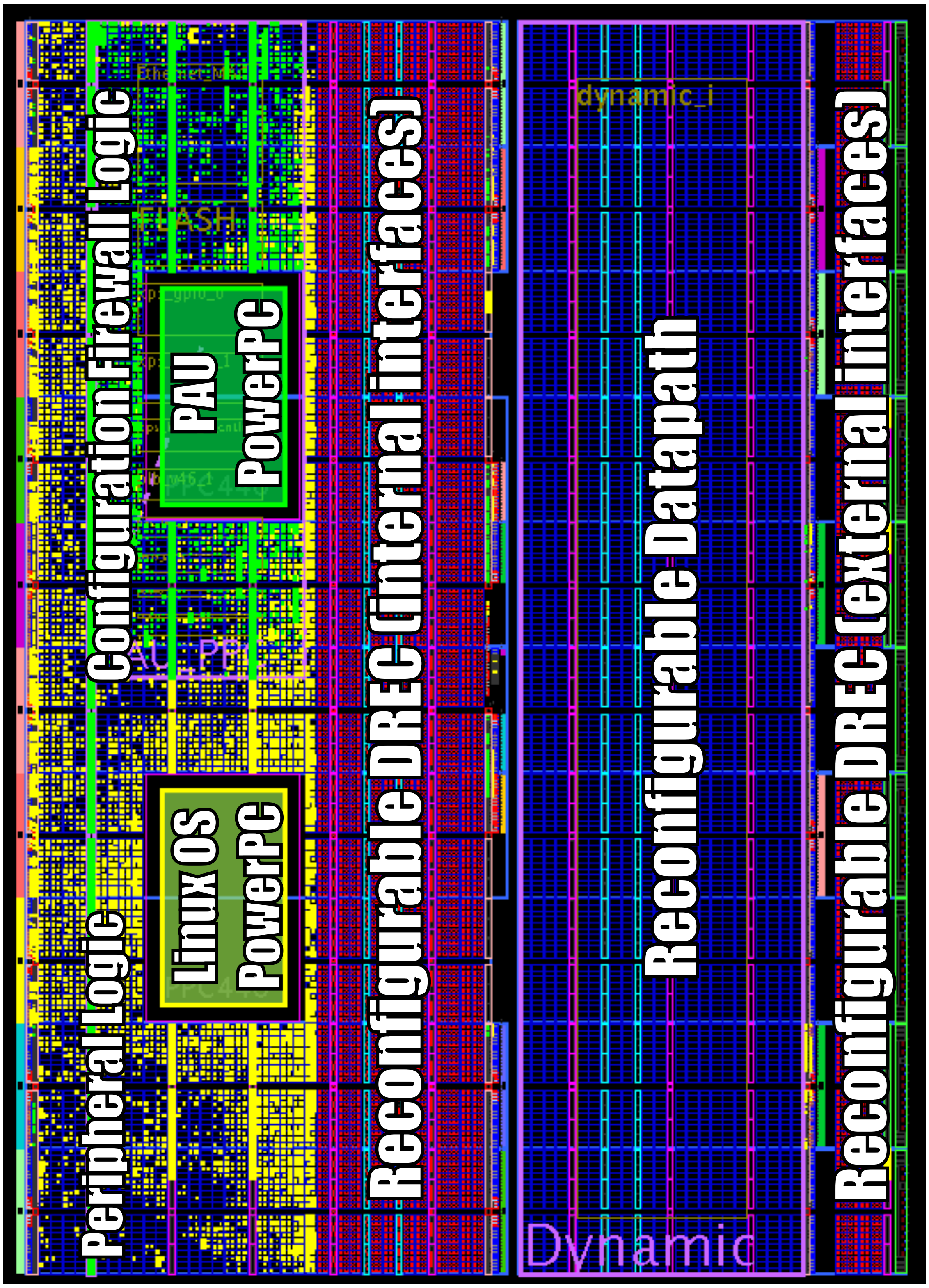

Peripheral Logic
Configuration Firewall Logic
PAU
PowerPC
Linux OS
PowerPC
Reconfigurable DREC (internal interfaces)
Reconfigurable Datapath
Reconfigurable DREC (external interfaces)
dynamic_i
Dynamic
FLASH

Figure 6.3: CHARE floorplan on a Xilinx Virtex-5 FX130T FPGA

The CHARE platform has a configuration firewall containing a dedicated Plug-in Assist Unit (PAU) processor for secure communication with the DMS. Similar to the protocols and safeguards used in cryptographic co-processors and TPMs, the configuration firewall performs critical functions in isolated hardware and does not share processor, logic, memory or routing resources with other CHARE subsystems. A private key embedded within the PAU provides a public key-based authentication protocol with the DMS. Decrypted partial FPGA configurations for hardware plug-ins are not revealed outside the configuration firewall or even to PAU software. External flash memory stores encrypted partial bitstreams received from the DMS, with just-in-time decryption of bitstreams transferred to the FPGA's Internal Configuration Access Port (ICAP). For the sake of both speed and security, a hardware-implemented flash memory controller controls the ICAP.

6.5 A Cognitive Radio Example

The need for flexibility and better spectrum management gave impetus to the cognitive radio (CR). A CR is a smart software defined radio that adapts its configuration based on perceived changes in its environment [15]. By sensing the spectrum, a CR detects and leverages opportunities by tuning communication variables such as the waveform, transmission power, modulation scheme, and operation frequency. Reconfigurable hardware suits both the performance and flexibility required by a CR [16]. To ensure integrity of the shared spectrum resource, the CHARE platform provides hardware oversight of physical-layer radio policies.

A simple digital AM transceiver with variable parameters is selected to demonstrate the application of CHARE to a CR platform. The AM transceiver architecture shown in Fig. 6.4 implements:

$$Y(t) = A_0 S(t) \cos(\omega_0 t + \phi), \tag{6.1}$$

where carrier frequency ω_0 and gain A_0, the two transmission parameters, are updated by issuing software update requests based on the radio's interaction with its environment. Xilinx's System Generator for DSP is used to construct the datapath. The carrier frequency is adjusted by modifying a Direct Digital Synthesizer (DDS) core parameter, while gain is adjusted by modifying an input to a multiplier core.

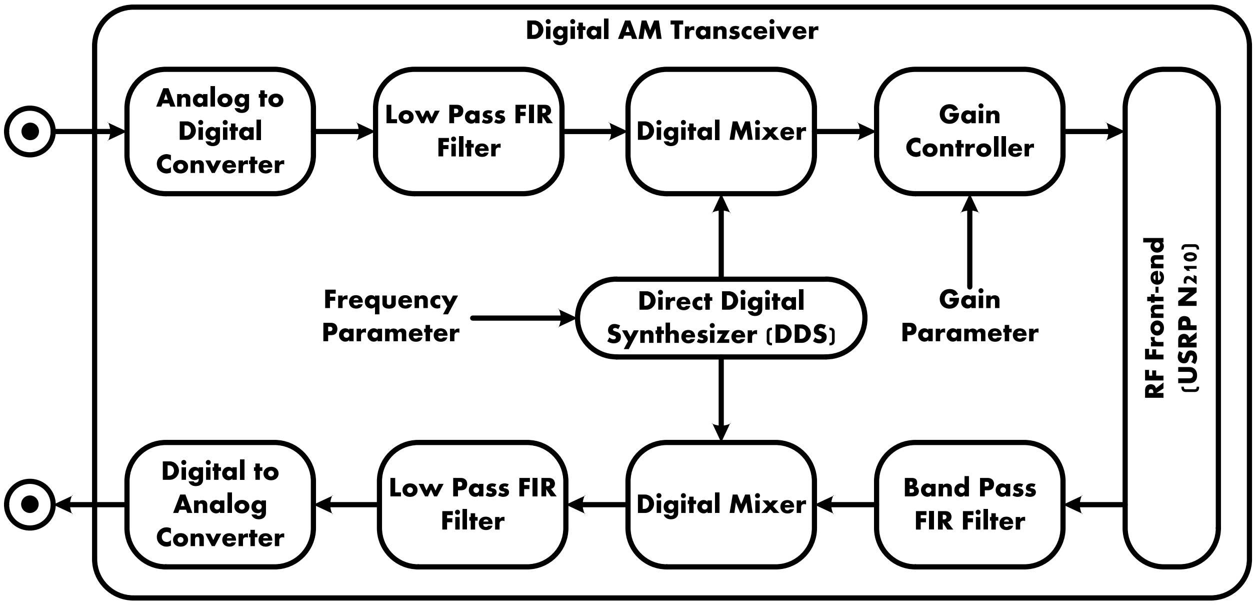


Figure 6.4: Digital AM transceiver architecture.

6.5.1 Policy Synthesis and Enforcement

Policies are derived from specifications of expected system behaviors. Specifications may describe, for example, expected software or hardware interactions with physical processes. Modern trust solutions fail to ensure that software functions, such as the embedded cognitive engine program in this CR example, conform to the policies in all instances without introducing additional, unknown behaviors. The CHARE platform demonstrates that policies can, however, be compiled into a hardware root-of-trust that is tightly integrated with a datapath to prevent undesired outside influence and erroneous internal behaviors.

In the CR example, configuration update requests are issued by the software-based cognitive engine to the reconfigurable hardware datapath. Policies are used to describe specifications that the datapath must always conform to, regardless of outside influence. Datapath update requests are specified as a set of registers which store update values for the DREC to evaluate against independently supplied policies, and update procedures which instruct the DREC on how to apply the updates to the datapath. Policies are compiled into satisfiability groupings, which are inferred from policy types. These groupings validate update requests and perform online monitoring of the datapath. When all policies are satisfied, the DREC performs the datapath update procedure associated with an update request.

Table 6.1: Policies with corresponding update registers.

Policy	Type/Grouping	Frequency Range (MHz)	Maximum Gain (dBm)	Time Range
`P1`	`Allow/G1`	`2.0-3.0`	`-10`	`0100-0200`
`P2`	`Allow/G1`	`2.0-3.0`	`-15`	`0300-0400`
`P3`	`Allow/G1`	`2.0-3.0`	`-10`	`0500-0600`
`P4`	`Allow/G1`	`3.5-4.5`	`-10`	`0700-0800`
`P5`	`Allow/G1`	`3.5-4.5`	`-15`	`0900-1000`
`P6`	`Monitor/G2`	`-`	`-`	`0000-1800`

Table 6.1 illustrates policies applied to a CR datapath. The policies are initially specified as a collection of the signal frequency and gain updates that can be made at certain times. Policies are typically associated with datapath update registers, parameters, or signals. Registers corresponding to policies `P1` through `P6` are shown in the rightmost three columns of Table 6.1. The CHARE compiler synthesizes policy specifications into declarative assertions in the form of Bluespec SystemVerilog (BSV) rules that make up a policy set within the body of the DREC [4]. Targeting BSV enables efficient and verifiable generation of hardware transactors. BSV's atomicity and expressivity also aid policy specification, and provide abstractions familiar to software-oriented datapath designers. Atomicity ensures that rules are entirely executed as they are always enabled in the DREC. The CHARE compiler leverages Bluespec to manage potential interactions between rules through the automatic generation of arbitration and scheduling logic.

The CHARE compiler translates registers associated with a datapath update request into a BSV action method enabled when the request occurs. Any number of update requests may be used with all or a subset of the datapath registers. In this application, a single update request named `UR1` is specified which requires the DREC to check the entire set of datapath registers for policy conformance, as seen in Fig. 6.5. `UR1` is used to send updates to the `freq` and `gain` datapath update registers.

Enabling the update method causes the DREC controller to evaluate the datapath register values corresponding to frequency, gain, and time against the policies `P1` through `P6`. Because update requests are not linked directly to policies, every request must conform to all groupings of policy rules. Unlike DARPA's neXt Generation platform, policies cannot be added, removed, or deactivated without performing a complete DREC hardware replacement [7]. For applications where only a subset of policies are active at a given time, the remote DMS issues new DREC-wrapped datapath plug-ins.

```
update_request(UR1)

update_registers(FixedPoint#(0,16) freq,
                 Int#(32) gain,
                 UInt#(12) time)

update_procedure(UP1)
```

Figure 6.5: Update request format.

Policies P1 through P5 are all of type Allow and are grouped into a single satisfiability BSV rule G1 that disables further DREC processing of the update request if none of the policies are satisfied. Update requests that do not satisfy any of the Allow policies are always rejected. The DREC can provide feedback as to which policies are not satisfied in a given update request, which may be useful for software learning or strategy development. Policies may also be hidden from software generating the update requests, in which case only a DREC accept or deny response is returned.

Policies can also act as monitors of datapath update registers and datapath parameters. Policy P6 is specified as type Monitor and is used to observe the time register. P6 asserts that the time register will always fall within 0000 to 1800. When the time falls outside of this range, the policy's satisfiability, G2, is no longer met and a response action is triggered in the DREC. This action could be used to disable datapath modules for various reasons, for example to save power in certain sections of a satellite's orbit, to enforce a mandatory offline built-in self-test cycle, or to expire IP after some period of system operation.

Monitors can also be placed on module interfaces or internal signals when specified as continuous or dynamic assertions. Monitors may be used in this manner to provide assurances on untrusted or poorly understood IP interfaces. Continuous assertions are specified directly in the DREC module and checked at each clock cycle. Similar to policies on datapath update registers, dynamic assertions are compiled to BSV rules in the DREC and evaluated whenever the rules are enabled. An optional guard may be placed on a rule containing dynamic assertion logic causing the rule to be enabled only when some condition is met [4, 20].

The CHARE compiler can also be extended to support commonly used libraries of assertions. For example, assertions found in the Acellera Standard Open Verification Library (OVL) are compiled to the DREC using the BSV interfaces and wrappers found in the OVLAssertions package [1]. Incorporating such monitors increases run-time observability which helps alleviate design verification and fault tolerance concerns.

Update requests are linked to update procedures that perform datapath plug-in reconfigurations or parameter adjustments on dynamically reconfigurable modules. When a plug-in reconfiguration is requested, the update procedure is written simply as an update code that the DREC passes to the PAU. If a datapath module parameter update is performed, the update procedure is written as the succession of values that the DREC must apply to registers representing datapath parameter values. The CHARE compiler synthesizes these procedures into sequences of BSV actions. In this application, a datapath parameter update procedure `UP1` is associated with update request `UR1`. `UP1` is specified as a series of parameter values that are written in order to phase shift the datapath's DDS modules and apply a new coefficient to the gain controller. The datapath parameter values written in `UP1` do not correspond directly to datapath update registers used in the request and are not visible to the source of the request. Simultaneous, non-conflicting update procedures are parallelized in the DREC controller through the use of BSV `par` blocks.

Integration of the DREC with the underlying datapath is achieved with the Bluespec ImportBSV package. The DREC wraps the datapath, either applying procedures to interface signals or simply passing them through to data buffers or external interfaces. Applying policies on all datapath interfaces helps to ensure external devices cannot cause datapath updates. The Bluespec compiler is then used to generate RTL Verilog containing the original datapath and integrated DREC. A black-box version of the datapath is used during compilation to reduce implementation time and ensure Bluespec does not introduce internal modifications. The generated RTL and original datapath are then processed by the PATIS and ISE PR implementation flow to produce a device programming bitstream. Any logic overhead incurred using BSV is acceptable for many applications as the resultant implementation is often more timing efficient than what would be produced using conventional HDL [2, 22].

6.5.2 Operational Verification

A trustworthy security solution requires rigorous verification to ensure defined and correct behavior in all situations. Rather than rely solely on functional simulation to validate a solution, formal verification is also used to provide a correctness proof. The two main elements of formal verification are an accurate design model and precise specifications of required properties.

Operation of CHARE subsystems are formally verified through both policy-dependent and policy-independent property checking. This is accomplished using compositional model checking, which relies on a simple set of proof techniques and a domain-specific strategy [17]. The goal of this strategy is to reduce the verification of a large system to smaller and more tractable sub-modules. A proof system supporting this approach generates verification subgoals discharged by Cadence's SMV symbolic model checker. In such a verification framework, the DREC is modeled as a finite state machine with an access state corresponding to the BSV update request action method. The access state directly enforces spectrum access policies given in Table 6.1 by responding to an update request `UR1` and checking its conformance to policies. An update response is specified as a propositional formula capturing policies `P1` through `P6`. The update procedure `UP1` is granted for requests satisfying policy constraint predicates.

The DREC security specifications include safety and liveness properties. Liveness properties ensure a response for every update request occurs without deadlocks or livelocks, whereas safety properties guarantee that policies are correctly applied [9]. All datapath update requests should eventually have a response, an update request that complies with policy rules should be granted access, and an update request that does not comply with policy rules should be denied. These security properties have the following form in first-order temporal logic:

$$\begin{aligned}
&\textbf{always}\,(\mathsf{request} \rightarrow \textbf{eventually}\,\mathsf{respond}),\\
&\textbf{always}\,(\mathsf{valid\_request} \rightarrow \textbf{eventually}\,\mathsf{grant}),\\
&\textbf{always}\,(\mathsf{invalid\_request} \rightarrow \textbf{immediately}\,\mathsf{deny}).
\end{aligned}$$

6.6 Conclusions

Most of the major advances in computer system security rely on hardware for enforcement. Data-intensive embedded platforms may require a hardware reorganization capability, introducing development complications and new types of security threats including zero-day attacks. CHARE automatically generates application-specific dynamic hardware plug-ins consisting of controller-wrapped datapaths. Software drivers are simplified by the controller's encapsulation and abstraction of the datapath structure. Hardware augments software monitoring and enforcement through the synthesis and formal verification of a controller incorporating datapath policy rules. As a result, CHARE simultaneously addresses the performance, power, developer productivity, and security requirements of high-throughput, reconfigurable platforms.

Acknowledgments

This work is part of the AUSTIN project supported by the National Science Foundation, Grant No. CNS-0910557.

References

[1] Accellera, Inc. *Accellera Standard OVL V2 Library Reference Manual*, July 2010. http://www.accellera.org.

[2] Abhinav Agarwal. Comparison of high-level design methodologies for algorithmic IPs: Bluespec and C-based synthesis. Master's thesis, MIT, Feb 2009.

[3] ARM. ARM security technology: Building a secure system using TrustZone technology, July 1999.

[4] Bluespec, Inc. *Bluespec SystemVerilog Reference Guide*, October 2009. http://www.bluespec.com.

[5] A. Chandrasekharan, S. Rajagopalan, G. Subbarayan, T. Frangieh, Y. Iskander, S. Craven, and C. Patterson. Accelerating FPGA development through the automatic parallel application of standard implementation tools. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 53–60, December 2010.

[6] Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, and Marko Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 15–20, 2007.

[7] Daniel Elenius, Grit Denker, and David Wilkins. XG Policy Architecture, April 2007. http://www.csl.sri.com/projects/xg/.

[8] Benjamin Glas, Alexander Klimm, Oliver Sander, Klaus Müller-Glaser, and Jürgen Becker. A system architecture for reconfigurable trusted platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 541–544, 2008.

[9] Vincent Hu, Richard Kuhn, Tao Xie, and JeeHyun Hwang. Model checking for verification of mandatory access control models and properties. *International Journal of Software Engineering and Knowledge Engineering*, 2010.

[10] T. Huffmire, B. Brotherton, Gang Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Security and Privacy, IEEE Symposium on*, pages 281–295, May 2007.

[11] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White, Ryan Kastner, and Tim Sherwood. Designing secure systems on reconfigurable hardware. *ACM Trans. Des. Autom. Electron. Syst.*, 13:44:1–44:24, July 2008.

[12] Ted Huffmire, Timothy Sherwood, Ryan Kastner, and Timothy Levin. Enforcing memory policy specifications in reconfigurable hardware. *Computers and Security*, 27(5-6):197–215, 2008.

[13] Intel Corp. Intel Trusted Execution Technology Overview.

[14] Intel Corp. Intel Virtualization Technology Overview.

[15] Ying-Chang Liang, Hsiao-Hwa Chen, J. Mitola, P. Mahonen, R. Kohno, J.H. Reed, and L. Milstein. Guest Editorial - Cognitive Radio: Theory and Application. *Selected Areas in Communications, IEEE Journal on*, 26(1):1–4, January 2008.

[16] Jorg Lotze, Suhaib A. Fahmy, Juanjo Noguera, Linda Doyle, and Robert Esser. An FPGA-based cognitive radio framework. In *Signals and Systems Conference (ISSC 2008)*, pages 138 –143, June 2008.

[17] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37:279–309, May 2000.

[18] Cameron Patterson. High performance DES encryption in Virtex FPGAs using JBits. In *8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 113–121, April 2000.

[19] Marcus Peinado, Yuqun Chen, Paul England, and John Manferdelli. NGSCB: A Trusted Open System. In *Information Security and Privacy*, volume 3108 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 2004.

[20] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'05)*, pages 15–24, 2005.

[21] Seth Schoen. Trusted Computing: Promise and Risk, October 2003. http://www.eff.org/wp/trusted-computing-promise-and-risk.

[22] Jiri Simsa and Satnam Singh. Designing hardware with dynamic memory abstraction. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 69–72, 2010.

[23] Trusted Computing Group, 2010. http://www.trustedcomputinggroup.org.

Chapter 7

Conclusions

7.1 Contributions

Embedded CPCs are increasingly penetrated in cyber attacks leading to disruptions in process operations and even physical damage. Unfortunately, tailored security at the CPC leaf nodes, i.e. embedded systems, is lacking for this domain. This dissertation work developed TAIGA, an architecture and design methodology for trustworthy embedded computing in CPCs. TAIGA is an autonomic, isolated, hardware-implemented, verifiable, tailored trustworthy space which monitors embedded controller interactions with external CPC components to ensure system-level specifications of stability and security are maintained at run-time. TAIGA protects physical processes by enforcing specification guards which are derived directly from a system's stability and security specifications. TAIGA can also predict and preempt unsanctioned controller behavior by leveraging precise models developed for physical processes.

TAIGA is an example of a fine-grained architecture that addresses five requirements of trusted components:

TR1 *The source code and implementation for the entire component is analyzed.* TAIGA is designed and implemented using high-level code, formal verification, and hardware synthesis tools. Since protection logic is tailored to the application, and only TAIGA additions need to be verified to assure security, the verification space is reduced to a manageable amount.

TR2 *The component uses private hardware resources for computation, internal communication, and memory, and does not invoke external components as subfunctions.* TAIGA's trusted components are self-contained and implemented entirely in isolated, programmable hardware. TAIGA is situated on the external interfaces of an embedded controller-under-protection to observe and potentially override all inputs and outputs to the controller.

TR3 *All external communication with untrusted components is through hardware-implemented, bounded, and isolated queues.* TAIGA uses opaque, fixed, hardware FIFOs to interface with embedded controllers and data from external units. TAIGA has full observability over its queues and can override their contained values as needed when specification guards are violated.

TR4 *The component cannot be bypassed or disabled, and has a fixed repertoire of essential services, such as I/O or cryptography.* TAIGA is autonomic and does not rely on an external components for operation or to preserve system stability and safety. TAIGA enforces specification guard logic which is derived from a system's stability and safety specifications and tailored to the application.

TR5 *Critical functionalities of the component, such as rule checking logic, cannot be updated without provably secure or physical access.* TAIGA is immutable at run-time outside of tunable specification guard values, which are bounded by system safety limits. TAIGA is implemented in isolated, programmable hardware, uses standard programmable logic device security mechanisms, and requires physical access to change rule checking logic.

The following is a listing of selected publications, funded proposals, and invited talks related to this work:

- **Lee W. Lerner**[1], Christopher J. McCarty, Kevin G. Lyn, and Cameron D. Patterson, "Trusting the Leaf Nodes: Embedded Security in Cyber-Physical Control," *Under Journal Review*, Submitted January 2015.
- Zane R. Franklin, Cameron D. Patterson, **Lee W. Lerner**[1], and Ron D. Prado, "Autonomic Hardware for Trust Enhancement of Critical Embedded Processes," *Proc. 7th International Symposium on Resilient Control Systems*, Denver, CO, Aug. 2014.

- **Lee W. Lerner**, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson, "Application-level Autonomic Hardware to Predict and Preempt Software Attacks on Industrial Control Systems," *Proc. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, GA, June 2014.

- **Lee W. Lerner**, Zane R. Franklin, William T. Baumann, and Cameron D. Patterson, "Using High-level Synthesis and Formal Analysis to Predict and Preempt Attacks on Industrial Control Systems," *Proc. 22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2014)*, Monterey, CA, Feb. 2014.

- **Lee W. Lerner**[1], "Hardware-oriented Trust Anchors for Embedded and Cyber-physical Systems", invited talk to Georgia Tech Research Institute, Atlanta, GA, Feb. 2013.

- **Lee W. Lerner**, "Trust for Cyber-physical Embedded Systems", invited talk to Xilinx Security Working Group (XSWG), Longmont, CO, Sept. 2012.

- Cameron D. Patterson, William Baumann, **Lee W. Lerner**, and Mohammed M. Farag, "Run-time Prediction and Preemption of Stuxnet-like Attacks in Embedded Process Controllers," NSF 12-503 Secure and Trustworthy Cyberspace (SaTC), Virginia Tech, 2012.

- **Lee W. Lerner**, Mohammed M. Farag, and Cameron D. Patterson, "Run-time Prediction and Preemption of Configuration Attacks on Embedded Process Controllers," *Proc. 1st International Conference on Security of Internet of Things (SecurIT 2012)*, Kerala, India, Aug. 2012.

- Jonathan P. Graf, Scott H. Harper, and **Lee W. Lerner**[2], "Ensuring Design Integrity through Analysis of FPGA Bitstreams and IP Cores," *Proc. The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'12)*, Las Vegas, NV, July 2012.

- Mohammed M. Farag, **Lee W. Lerner**, and Cameron D. Patterson, "Interacting with Hardware Trojans Over a Network," *Proc. 2012 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2012)*, San Francisco, CA, June 2012.

- Jonathan Graf, Scott Harper, and **Lee Lerner**[2], "Ensuring Design Integrity through Analysis of FPGA Bitstreams and IP Cores," *Government Microcircuit Applications and Critical Technology Conference (GOMACTech-12)*, Las Vegas, NV, March 2012.

- Mohammed M. Farag, **Lee W. Lerner**, and Cameron D. Patterson, "Thwarting Software Attacks on Data-Intensive Platforms with Configurable Hardware-Assisted Application Rule Enforcement," *Proc. 21st International Conference on Field Programmable Logic and Applications (FPL'11)*, Chania, Crete, Greece, Sept. 2011.

- **Lee W. Lerner**, "Configurable Hardware-Assisted Application Rule Enforcement (CHARE) for Data-Intensive Platforms," invited talk to CISO and Chief Security Architect of General Electric, Blacksburg, VA, July 2011.

7.2 Future Work

We have identified some current limitations with TAIGA and our proposed design methodology and architecture:

- High-level hardware synthesis tools are marketed as being nearly as easy to work with as source code compilers. However, significant learning and hardware debugging is typically required.

- The formal verification approach outlined which uses Frama-C also requires significant manual intervention and expertise.

- The recommended target TAIGA implementation technology, configurable SoCs, currently has a cost premium over standard microcontrollers. However, this may easily be eclipsed by the value of the physical process in many CPCs.

- The plant state preview window is ultimately limited by processing power and the need to periodically synchronize the model with the physical plant to keep up with changes due to disturbances or commands.

- Prediction may not be possible if process state also depends on events outside the model.

- While the inability to remotely update TAIGA guard logic is a security asset, the need for physical access to modify the model, process specifications, and backup controller incurs a higher maintenance cost. However, such updates are rarely needed.

[1]Performed in cooperation with Georgia Tech Research Institute

[2]Performed in cooperation with Luna Innovations Incorporated

- Architectural modifications may be required to scale and distribute TAIGA among multiple embedded systems in a complex CPC environment with many control loops.

Potential improvements can be made to TAIGA through the use of offline computations to make online TAIGA monitoring more efficient. To improve switchover logic we are investigating a neural network-based classifier to make intelligent decisions in real-time based on current sensor measurements. The functional purpose of the switchover mechanism is to decide when to switch from the production to backup controller and can be described as a classification problem. A classifier can be constructed using a multilayer perceptron as a feed-forward artificial neural network model that maps a set of inputs to outputs using a nonlinear activation function. Back propagation can be used to calculate a gradient of the loss function with respect to all the weights of the neural network followed by an optimization method to reduce the loss function [4]. Plant states are the classifier input and the data set for training the neural net is collected by applying the backup controller to a model of the plant for different sets of initial conditions corresponding to the current state. An algorithm can then be developed to monitor the plant states and check if transitioning to the backup controller at that instant can be done without violating specification guards.

Another area of ongoing research is TAIGA implementation and efficiencies in a variety of CPC environments. For instance, we are currently pursuing TAIGA integration with a rotary pendulum at Virginia Tech and with a Zynq-based motor controller running in Georgia Tech Research Institute (GTRI)'s ICS security test bed [1]. We are also investigating alternatives to implementing TAIGA during the controller development process, such as addressing how TAIGA integrates with existing, third-party CPC controllers. Alternatives to programmable logic-based TAIGA implementations might be more appropriate in some cases. Therefore software-based implementations might also be investigated, including a rigorous evaluation of corresponding efficiency and security.

In addition to making further architectural improvements to TAIGA and continual analysis of effectiveness, one final area of investigation is the possibility of applying TAIGA to other domains outside of run-time CPC protections. For instance, TAIGA also has a potential system development and analysis use by providing real-time interface observability and controllability. The ability to inject data or events and monitor reactions on interfaces facilitates analysis and test of such assertions under a variety of hypothesized scenarios, and without modifying the system components under test. This approach also permits investigation of sophisticated attacks exploiting unusual software and hardware interactions such as a reset or interrupt during a critical system operation. Multiple TAIGA layerings permit concurrent, real-time evaluation of attacks and defenses.

References

[1] Avnet. Xilinx Zynq-7000 All Programmable SoC/Analog Devices Intelligent Drives Kit, 2013. Avnet product brief.

[2] N. Falliere, L. O'Murchu, and E. Chien. W32.stuxnet dossier, 2011.

[3] Claude Marché and Yannick Moy. *The Jessie plugin for Deductive Verification in Frama-C*. INRIA Saclay - Télé-de-France and LRI, CNRS UMR 8623, 2013. `http://proval.lri.fr/`.

[4] M. Syiam, H. Klash, I. Mahmoud, and S. Haggag. Hardware implementation of neural network on FPGA for accidents diagnosis of the multi-purpose research reactor of Egypt. In *Proceedings of the 15th International Conference on Microelectronics*, ICM 2003, page 326329, 2003.

[5] Mark Zeller. Myth or reality – does the Aurora vulnerability pose a risk to my generator? In *Protective Relay Engineers, 2011 64th Annual Conference for*, pages 130–136. IEEE, 2011.

Appendix A

```
#pragma JessieFloatModel(defensive)      // Comment out for HLS flow

static const float C        = -0.7931f; // Derivative constant
static const float Gd       = 6.0324f;  // Derivative gain
static const float Gp       = 35.3675f; // Proportional gain
static const float Gi       = 0.5112f;  // Integration gain
static const float clip_min = -64.0f;   // Minimum controller output
static const float clip_max = 63.0f;    // Maximum controller output
static const float y_min    = -3.2f;    // Minimum allowed sensor input
    from a correctly functioning plant
static const float y_max    = 3.2f;     // Maximum allowed sensor input
    from a correctly functioning plant
static const float w_safe   = 0.0f;     // Quiescent reference
   controller input

static int reset = 0;   // Global state reset: 0 = inactive; 1 = active

// Returns argument (din) clipped to the range [clip_min, clip_max]
// Implemented in both hardware and software
/*@
    requires clip_min <= clip_max;
    assigns \nothing;
    behavior verify_clip:        // Verify clip() with Jessie
        ensures \result >= clip_min && \result <= clip_max;
*/
static float clip(float din)
{
```

```
//#pragma HLS allocation instances=Fcmp limit=1 core     // Comment out
   for Frama-C analysis

    float dout = din;

    if (din < clip_min)
    {
        dout = clip_min;
    }
    else if (din > clip_max)
    {
        dout = clip_max;
    }

    return dout;
}


// Production, software-implemented, PID controller
// Has latent, malicious behavior
// Inputs are the reference signal (w) and closed loop signal (y)
// Output is PID controller output (u)
/*
static float sw_controller(float w, float y)
{
    static float prev_x1, prev_x2, prev_yd, prev_yi;     // Previous PID
        states: X1(n-1), X2(n-1), Yd(n-1), Yi(n-1)
    static int cycle_count = 0; // Enables latent behavior

    if (reset)
    {
        prev_yi = 0.0f; // Reset Integrator stage
        prev_x2 = 0.0f;
        prev_yd = 0.0f; // Reset Derivative stage
        prev_x1 = 0.0f;
    }
    float e = clip(w - y);       // Error signal
    float yp = Gp * e;  // Proportional stage
    float x2 = Gi * e;  // Integrator stage
    float yi = clip(prev_yi + prev_x2 + x2);
    float x1 = Gd * e;  // Derivative stage
    float yd = x1 - prev_x1 - (C * prev_yd);

    prev_x2 = x2;        // Update internal PID states for the next
       iteration
    prev_yi = yi;
    prev_x1 = x1;
    prev_yd = yd;

    float u = clip(yp + yi + yd);        // P + I + D

    cycle_count++;

    return ((cycle_count < 100) ? u : clip_max);         // Malicious
       behavior
}
*/


// Backup, stability-preserving, hardware-implemented, proportional
   controller with a constant reference input
// Input is closed loop signal (y)
// Output is the controller output (u)
static float hw_controller(float y)
{
//#pragma HLS allocation instances=fAddSub limit=1 core // Comment out
   for Frama-C analysis

    float e = clip(w_safe - y); // Error signal
    float yp = Gp * e;  // Proportional stage
    float u = clip(yp);
    return u;   // Clipped output == (Gp * (w_safe - y)) ||  (Gp *
       clip_min) || (Gp * clip_max) || clip_min || clip_max
}


// Plant model, implemented in hardware
// Input is the controller output (u)
// Output is the plant's sensor output (y)
static float hw_plant_model(float u)
{
//#pragma HLS allocation instances=fmul limit=1 core    // Comment for
   Frama-C analysis

    static float y_z2, y_z1, u_z2, u_z1;        // Previous plant
       states

    if (reset)
    {
        y_z1 = 0.0f;    // Reset plant state
        y_z2 = 0.0f;
        u_z1 = 0.0f;
        u_z2 = 0.0f;
    }

    float y = (1.903f * y_z1) - (0.9048f * y_z2) + (0.0000238f * u) +
       (0.0000476f * u_z1) + (0.0000238f * u_z2);

    y_z2 = y_z1;        // Update internal plant states for the next
       iteration
    y_z1 = y;
    u_z2 = u_z1;
    u_z1 = u;

    return y;
}
```

```
// Returns whether the plant is operating within specifications, and
   implemented in hardware
// Input is the plant's sensor output (y)
// Output: 0 = false; 1 = true
/*@
    assigns \result;
    behavior verify_y_invalid:
        assumes y < y_min || y > y_max;
        ensures \result == 0;
    behavior verify_y_valid:
        assumes y_min <= y <= y_max;
        ensures \result == 1;
    disjoint behaviors verify_y_invalid,verify_y_valid;
    complete behaviors verify_y_invalid,verify_y_valid;
*/
static int hw_spec_guard(float y)
{
    return ((y >= y_min) && (y <= y_max));
}


// Hardware-implemented monitor (switchover control logic using the
   plant model, specification guards, and backup controller)
// Data inputs are the production controller's output (u_sw) and the
   physical plant's sensor output (y_physical)
// Return's either the production or backup controller's output (u)
// Assertions should confirm the subset of control code that remains
   active (the backup controller) when a process parameter goes out of
   spec
//@ ghost float ghost_y_model = 0.0f;
//@ ghost int   ghost_backup = 0;
//@ ghost float ghost_u_hw = 0.0f;
/*@
    assigns \nothing;
    behavior verify_all_valid:
        assumes y_physical >= y_min     && y_physical <= y_max     &&
                ghost_y_model >= y_min  && ghost_y_model <= y_max  &&
                y_accel >= y_min        && y_accel <= y_max;
        ensures ghost_backup == 0;
    behavior verify_any_invalid:
        assumes y_physical < y_min      || y_physical > y_max      ||
                ghost_y_model < y_min   || ghost_y_model > y_max   ||
                y_accel < y_min         || y_accel > y_max;
        ensures \result == ghost_u_hw;
    disjoint behaviors;
    complete behaviors;
*/
#define BACKUP_HOLD_COUNT 200
float hw_monitor(float u_sw, float y_physical, float y_accel)
{
    static int   backup_hold;
    static float y_model;

    if (reset) y_model = hw_plant_model(u_sw);

    float u_hw = hw_controller(y_physical);
    //@ ghost y_model = ghost_y_model;  // Assignment here allows
       y_model reset
    int backup = !hw_spec_guard(y_physical) ||
                 !hw_spec_guard(y_model)    ||
                 !hw_spec_guard(y_accel);

    backup_hold = (backup) ? (BACKUP_HOLD_COUNT) : (backup_hold - 1);
    backup_hold = (backup_hold < 0) ? 0 : backup_hold;
    float u = (!backup && !backup_hold) ? u_sw : u_hw;
    y_model = hw_plant_model(u);
    //@ ghost ghost_backup = backup;
    //@ ghost ghost_u_hw = u_hw;
    return u;
}
```