Using High-level Synthesis to Predict and Preempt Attacks on Industrial Control Systems

Zane Ryan Franklin

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

Master of Science in Computer Engineering

Cameron D. Patterson, Chair
William T. Baumann
Patrick R. Schaumont

March 31, 2014
Blacksburg, Virginia

Keywords: Industrial control systems, security, reconfigurable platform, high-level synthesis, system-on-chip

Copyright 2014, Zane Ryan Franklin
Using High-level Synthesis to Predict and Preempt Attacks on Industrial Control Systems

Zane Ryan Franklin

ABSTRACT

As the rate and severity of malicious software attacks have escalated, industrial control systems (ICSes) have emerged as a particularly vulnerable target. ICSes govern the automation of the physical processes in industries such as power, water, oil and manufacturing. In contrast to the personal computing space, where attackers attempt to capture information or computing resources, the attacks directed at ICSes aim to degrade or destroy the physical processes or plants maintained by the ICS. Exploits with potentially catastrophic results are sold on brokerages to any interested party.

Previous efforts in ICS security implicitly and mistakenly trust internal software. This thesis presents an architecture for trust enhancement of critical embedded processes (TECEP). TECEP assumes that all software can be or has already been compromised. Trust is instead placed in hardware that is invisible to any malicious software. Software processes critical for stable operation are duplicated in hardware, along with a supervisory process to monitor the behavior of the plant. Furthermore, a copy of the software and a model of the plant are implemented in hardware in order to estimate the system’s future behavior.

In the event of an attack, the hardware can successfully identify the plant’s abnormal behavior in either the present or the future and supersede the software’s directives, allowing the plant to continue functioning correctly. This approach to ICS security can be retrofitted to existing ICSes, has minimal impact on the ICS design process, and modestly increases hardware requirements in a programmable system-on-chip.
Acknowledgments

Thanks to my advisor, Dr. Cameron Patterson, for the opportunity to be a part of this project and for his advice on academics, research and life in general. Thanks to Dr. William Baumann and Dr. Patrick Schaumont for their participation on my advisory committee. Thanks to Lee Lerner for encouraging me to join this project and for the opportunity to further explore its real-world applications.

Thanks to my family for their support and guidance. A special thanks to my lovely wife, Vivian, for her unceasing love and encouragement. Thanks to the friends I met in my time at Virginia Tech, who made Blacksburg a place I will remember fondly. Finally, thanks to El Elyon: the King who was, and is, and is to come.

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 author and do not necessarily reflect the views of the National Science Foundation.

Zedboards and design tools were donated by Xilinx, Inc.
Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Motivation .............................................. 1
1.2 Contributions .......................................... 3
1.3 Thesis Organization ..................................... 4

2 Industrial Control Systems 5

2.1 Distributed Control Systems ........................... 5
  2.1.1 Master Terminal Unit ................................. 6
  2.1.2 Remote Terminal Unit ................................. 7
  2.1.3 Programmable Logic Controller .................. 7
2.2 Control Loops ............................................. 7
2.3 PID Controller ........................................... 8
2.4 ICS Security .............................................. 9
2.4.1 Threats ............................................................... 10
2.4.2 Countermeasures .................................................. 12
2.4.3 Assumptions and Limitations ................................... 14

3 Programmable System-on-Chip Overview ........................................... 16

3.1 Target Platform .......................................................... 17
3.1.1 Zynq Processing System ........................................... 17
3.1.2 Programmable Logic ................................................ 19

3.2 Programmable System-on-Chip Design ......................................... 21
3.2.1 Model-Based Design ................................................ 22
3.2.2 Hardware Development ............................................. 24
3.2.3 Software Development ............................................. 25

3.3 TECEP on a Programmable SoC ........................................... 26

4 TECEP Security System Overview ................................................. 28

4.1 Production Controller ................................................... 29

4.2 System Monitor .......................................................... 30
4.2.1 Backup Controller .................................................. 31
4.2.2 Specification Guards ............................................... 31
4.2.3 Control Logic ....................................................... 32
4.2.4 Plant Model ......................................................... 32

4.3 Prediction Unit .......................................................... 33
List of Figures

2.1 Distributed control system ........................................ 6
2.2 An example centrifuge control loop ............................. 7
2.3 Proportional-integral-derivative controller ....................... 9
2.4 Control loop threats .................................................. 11
2.5 Motor controller with TECEP ...................................... 13

3.1 Xilinx Zynq-7000 SoC Architecture ............................. 18
3.2 Abstract FPGA architecture ........................................ 21
3.3 Configurable logic block .............................................. 22
3.4 DSP slice .............................................................. 23
3.5 Block RAM ............................................................ 23
3.6 Programmable SoC design process ................................. 25
3.7 Model-based design flow ............................................. 27

4.1 System components .................................................. 29
4.2 TECEP component isolation ........................................ 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Prediction unit components</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Junction box component connections</td>
<td>35</td>
</tr>
<tr>
<td>4.5 Junction box flow of control</td>
<td>36</td>
</tr>
<tr>
<td>4.6 Trust in system components</td>
<td>38</td>
</tr>
<tr>
<td>5.1 Platform components</td>
<td>40</td>
</tr>
<tr>
<td>5.2 HLS interface adapters</td>
<td>44</td>
</tr>
<tr>
<td>5.3 Junction box internal control</td>
<td>45</td>
</tr>
<tr>
<td>5.4 Vivado block design</td>
<td>47</td>
</tr>
<tr>
<td>6.1 Plant output with no countermeasures</td>
<td>50</td>
</tr>
<tr>
<td>6.2 Plant output with TECEP additions</td>
<td>51</td>
</tr>
<tr>
<td>6.3 Plant output with TECEP additions and prediction</td>
<td>52</td>
</tr>
<tr>
<td>6.4 Plant output with return to production controller</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Tables

2.1 PID control algorithm terms ........................................ 10
2.2 ICS threats .............................................................. 12
2.3 Separating a SoC into a Normal Zone (NZ) and Secure Zone (SZ) .... 14
2.4 ICS threats .............................................................. 15
3.1 Zynq I/O controllers ................................................... 19
3.2 Zynq-7020 resources .................................................. 24
5.1 Motor control system parameters ....................................... 41
5.2 System monitor interface ............................................... 43
5.3 Prediction unit operations .............................................. 46
6.1 Motor Controller-specific attributes ..................................... 48
6.2 Programmable logic resource utilization .............................. 54
6.3 Component runtimes .................................................... 54
Chapter 1

Introduction

1.1 Motivation

Computer-related criminal activity is continually increasing in frequency and severity. The attention received from major news and media outlets is increasing in tandem. In opposition, cybersecurity has become a multi-billion dollar industry. Microsoft, Symantec, Kaspersky, and many other corporations produce numerous software suites at both the consumer and enterprise level. A multitude of companies have research divisions devoted to the development of new strategies and mechanisms for device and network security. In the U.S., agencies such as the FBI and NSA are taking increasingly drastic measures toward the prevention and punishment of cyber attacks [1]. Despite the rapid growth of the security industry, an all-time high in popular awareness, and escalated consequences for conviction, cybersecurity has been unable to keep up with the capabilities of cyber-criminals.

Rather than designing software with security as an objective, it is largely treated as an afterthought. As a result, many organizations rely heavily on perimeter security, such as firewalls and passwords. Over half the respondents to a recent survey of IT security professionals believe that their perimeter security is insufficient and may have already been
compromised [2]. Furthermore, perimeter security is completely ineffective against internal threats. There is little within a network to restrain an attack once it has bypassed the perimeter.

According to Eugene Spafford, a leading expert on cybersecurity, “firewalls were a stopgap measure until the [security mechanisms in] hosts were fixed.” [3]. Unfortunately, firewalls, along with patches, are now accepted as standard security measures. Patches aim to fix or remove a particular security vulnerability. Patching obviously requires that exploits be detected, which is often accomplished by a third party. Brokerages sell zero-day exploits to the highest bidder, whether it be the organization responsible for the software (in the interest of patching) or a rival institution with malicious intent [4].

System-level security for general-purpose computing is beyond the scope of this work. The versatility needed by for such devices complicates their security needs significantly. Additionally, a certain level of usability is expected while simultaneously placing a significant level of responsibility on the user. Therefore, this work will focus on application-specific computing platforms, and more specifically on industrial control systems. The term “industrial control system (ICS)” refers to the series of electronic devices used to automate the physical processes (plants) in industries such as power, oil and gas, water and manufacturing. ICSes are particularly vulnerable to attack, as their network connection is intended only to provide remote access to a supervisor, and is therefore lacking in security.

Although ICS perimeter security is weak, some viable internal security methods have been developed. Runtime monitoring incorporates a supervisor or peer device which monitors the behavior of the ICS and its plant at runtime and reports any anomaly to a human operator [5]. One disadvantage of this method is its reactive nature. Although an attack may be detected, it is nonetheless successful in negatively affecting the plant and requires corrective action. As an analogy, the knowledge that one’s bank account has been emptied is of course useful, but does not prevent the theft.

Equally if not more problematic is the instinctive trust placed in the monitoring software.
More generally, operating systems, drivers, communication protocols, and nearly all underlying software are inherently trusted in an ICS. If these trusted components can be compromised, no amount of runtime monitoring or even secure design methods can neutralize the threat of a Stuxnet-like attack.

1.2 Contributions

Therefore, this work proposes a trust enhancement of critical embedded processes (TECEP), which assumes that all software in a system can be or has already been compromised. Any process that is critical for stable operation is duplicated in autonomic hardware. Autonomic hardware is transparent to any remote or local software and therefore beyond the reach of a software attack. The primary objective of implementing these processes in configurable hardware is security. Performance and power improvements are merely incidental benefits. ICSes are an ideal initial target for TECEP due to the strictly defined specifications which govern the proper behavior of the system, and the existence of accurate physical process models.

Monitoring and specification guard functions are captured in C, which permits C-based formal analysis and verification. Once this code has been formally verified, it is synthesized into configurable hardware using High-Level Synthesis (HLS). Also synthesized using HLS is a high-assurance backup controller running parallel to the primary controller.

An identical copy of the primary controller and model of the plant operate in the ghost hardware as well. This pair is used as a prediction unit to make inferences about the future behavior of the primary controller and the plant based on their past behavior and current inputs. Because they are isolated in hardware, the monitor and backup controller are inaccessible to malicious software and capable of preserving plant stability in the event of an attack. Furthermore, the prediction unit allows the system to anticipate an attack and act before its effects manifest. The monitor acts as the top-level module in the system, auditing
the outputs of the primary controller and prediction unit and ensuring they stay within an acceptable range. If either moves beyond this range, indicating a current or future error, the monitor directs the high-assurance backup controller to supersede the primary controller, allowing the plant to remain stable.

The security mechanisms proposed here can be smoothly integrated with the traditional methods of control system design without imposing additional responsibilities on the control system engineer. A formal analysis expert can develop and verify the monitor and specification guards based on the ICS parameters. A platform expert can integrate the TECEP components with the existing controller while recycling existing resources such as the control system engineer’s plant model. Extensive hardware expertise is not necessary due to the use of HLS in lieu of traditional hardware design methods. The hardware requirements introduced by TECEP coupled with the traditional requirement of an embedded processor lends itself well to a system-on-chip (SoC) design. This work therefore targets the Xilinx Zynq-7000 SoC platform.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 1 discusses the motivation and contributions of this work. Chapter 2 provides a description of industrial control systems, their threats models and existing security measures. Chapter 3 describes the target platform as well as the general system-on-chip design and implementation process. Chapter 4 includes a high-level discussion of the individual components in the system, while Chapter 5 provides the application-specific details of the system used in this work. Chapter 6 presents and evaluates the results of the given implementation. Finally, Chapter 7 summarizes the successful contributions of this thesis and outlines future work and improvements.
Chapter 2

Industrial Control Systems

Cyber-physical systems consolidate physical material or processes with electronic components. ICSes, a subset of cyber-physical systems, are widely employed across industries such as power, water, gas, oil and manufacturing. The fundamental element of an ICS is a control loop. Control loops consist of control algorithms running on some computing platform (often a microcontroller) that regulate some physical process (called the plant). This chapter provides a basic understanding of ICSes as well their contemporary attack vectors and countermeasures.

2.1 Distributed Control Systems

A distributed control system (DCS) is a hierarchy of control loops and supervisors distributed across multiple physical locations. Hundreds or even thousands of individual control loops may be assimilated within an ICS. Fig. 2.1 demonstrates a DCS network. The primary components of a DCS are master terminal units (MTUs), remote terminal units (RTUs) and programmable logic controllers (PLCs). MTUs exist at the top of the control hierarchy, collecting data and providing an interface to human operators. PLCs are included in control
loops with physical processes, and RTUs manage PLCs and mediate between PLCs and
MTUs. In recent years, some overlap has developed between these components and their
functions have become less distinct.

2.1.1 Master Terminal Unit

An MTU’s primary role is to provide an interface between the ICS and human operators.
MTUs are PCs running supervisory control and data acquisition (SCADA) software [6].
SCADA systems interact with RTUs, collecting data from remote locations and providing
control updates as necessary. An example SCADA implementation is a regional power grid.
MTUs at a central station monitor RTUs at regional power stations, which in turn interact
with other RTUs and PLCs at local power plants.
2.1.2 Remote Terminal Unit

Remote telemetry units originally acted as mediators between PLCs and MTUs [7]. A remote telemetry unit functioned only as an I/O device, passing updates to PLCs and collected data back to MTUs. This operation allowed human operators at a central location to access remote plants. More recently, remote terminal units (RTUs) have bridged the gap between remote telemetry units and PLCs [8]. RTUs now provide both control processes and the ability to relay data throughout the ICS hierarchy.

2.1.3 Programmable Logic Controller

A PLC is an embedded real-time system that is designed for multiple I/O provisions [9][10]. PLCs are hardened for use in industrial applications against factors like temperature, vibration and electrical noise. Although they are customizable, PLCs are generally programmed for specific automation tasks with periodic updates. However, for high-volume applications, PLCs increasingly incorporate or are replaced by microcontroller-based systems.

2.2 Control Loops

![An example centrifuge control loop](image.png)

Figure 2.2: An example centrifuge control loop
A control loop incorporates a plant and its controller, which may be a PLC or an RTU. Fig. 2.2 shows a simple closed control loop, in which feedback from the plant affects the controller’s commands. Here the plant is an electric motor driving a centrifuge. During each system cycle, sensors on the motor collect data such as speed and torque and relay it to the controller. The controller takes this data and its current state as inputs and produces a new command for the motor. This output is sent to an actuator, which converts the signals into electromechanical action.

As an example, there may be four such control loops in a vehicle (one per wheel). A supervisor would monitor the speed of each wheel and adjust the torque or braking applied to each wheel. This exchange could then be used to improve traction control. Multiple control loops are often deployed in this fashion in order to coordinate disparate elements of a physical system.

2.3 PID Controller

A proportional-integral-derivative (PID) controller is a simple control structure used in ICS applications. Fig. 2.3 models a PID controller, and Equation 2.1 computes its output [11].

\[ u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt + K_p \cdot \frac{de}{dt} \]  

(2.1)

Table 2.1 describes the variables of Equation 2.1. At any time \( t \), the current value of the plant is compared to an ideal reference value to compute the error margin \( e(t) \). This error is multiplied by the proportional gain \( (K_p \cdot e(t)) \) to compute the current error margin. A sum of past errors is computed by multiplying the integral of the error by the integral gain \( (K_i \cdot \int e(t)dt) \). Multiplying the derivative of the error by the derivative gain \( (K_p \cdot \frac{de}{dt}) \) produces the anticipated future error [12]. The output of the controller is the sum of these three terms.

In order to simplify computation, Equation 2.2 can be produced by taking the Laplace
Figure 2.3: Proportional-integral-derivative controller

The transform of Equation 2.1.

\[ u(s) = K_p + \frac{K_i}{s} + K_d \cdot s = \frac{K_d \cdot s^2 + K_p \cdot s + K_i}{s} \]  \hspace{1cm} (2.2)

Equation 2.2 can then be used to compute the controller output without the need for complex calculations of integration and differentiation.

### 2.4 ICS Security

As smart grids expand and the cyber-physical world becomes increasingly networked, cyber-attacks have become an increasing threat to ICSes compared to physical attacks [13]. ICSes are networked for the purpose of reliability, monitoring and maintenance; they are typically not designed with any form of security in mind [14]. Therefore the contemporary approach
Table 2.1: PID control algorithm terms

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_p$</td>
<td>Proportional Gain</td>
</tr>
<tr>
<td>$K_i$</td>
<td>Integral Gain</td>
</tr>
<tr>
<td>$K_d$</td>
<td>Derivative Gain</td>
</tr>
<tr>
<td>$e$</td>
<td>Error = Reference Input – Plant Output</td>
</tr>
<tr>
<td>$t$</td>
<td>Time</td>
</tr>
</tbody>
</table>

to security for a DCS (Fig. 2.1) is to treat it as a generic network. Perimeter security is prioritized with tools like firewalls and controlled access, and security flaws are patched with regular updates. Unfortunately, this approach is inadequate due to the responsibility of ICSes and particularly of PLCs.

2.4.1 Threats

Firewalls can be bypassed and access credentials falsified [15]. The inadequacy of firewalls was demonstrated recently with the U.S. power grid. While testing a communication protocol, an engineer discovered that he was able to gain unrestricted access to a power substation, which provided the potential to cause a sweeping power outage [16]. Because these protocols are necessary for SCADA operation, they must be given access to the system.

Once an attack has penetrated the perimeter, PLCs are particularly vulnerable, as they are often simple microcontrollers with no inherent defenses. However, these simple PLCs have the final authority in controlling the plant, along with the ability to degrade or even destroy it.

Latent malware within a control program was the alleged cause of the Trans-Siberian Pipeline’s destruction in 1982, the largest non-nuclear explosion ever seen from space [17][18]. Stuxnet was likely propagated to its intended target using infected USB drives, and had a detailed
understanding of the Siemens control software it targeted. It has been hypothesized that Stuxnet infected over 50,000 computers and likely delayed the commissioning of an Iranian nuclear plant [19]. Aurora [20] targeted communications and relays in electric power generators. An Aurora attack demonstration showed the potential to cause severe generator damage by simply desynchronizing the opening and closing of a breaker. This work therefore targets threats that exist within an ICS.

Fig. 2.4 highlights some of these threats in red. Regular software updates are distributed to the controller via the DCS hierarchy. These updates could contain malicious behaviors that attempt to destroy the plant while reporting normal behavior to a human operator. Data transferred between the controller and plant could be interrupted or falsified. This would allow normal controller behavior while damaging the plant. Data interruption/falsification is of concern especially when wireless radios are used for communication between components.

![Control loop threats](image)

**Figure 2.4: Control loop threats**

Attack vectors that require physical access such as side-channel analysis or fault-injection are beyond the scope of this work. It is assumed that ICSes are physically secure by means of walls, guards and surveillance. If physical access is available to the device, a more obvious and direct means of destruction would likely be used. Instead, the primary threat focus here is the inherent trust placed in control software, operating systems, communication protocols and drivers. Particularly of concern is the updatability of these components, which nullifies
design-time verification and opens the system to Stuxnet-like attacks.

Table 2.2 lists potential ICS threats and their requisite levels of access. The primary threats addressed in this work are those which do not require physical proximity. Maliciously updating software or compromising supervisory nodes may be accomplished remotely, with only network access required. Although interference outside the PLC is a notable threat, modifying sensor or actuator data requires physical access to the system. The threat model used for this work is similar to that of the emerging Internet of Things [21]. A cyberattack on embedded devices poses less risk and potentially greater reward than breaking into a home or business, and has therefore become a more prevalent threat [22].

<table>
<thead>
<tr>
<th>Threat</th>
<th>Access Required</th>
<th>Threat Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious software updates</td>
<td>Remote update mechanism</td>
<td>High</td>
</tr>
<tr>
<td>Malicious supervisory node</td>
<td>Network</td>
<td>High</td>
</tr>
<tr>
<td>Sensor interference</td>
<td>Physical</td>
<td>Moderate</td>
</tr>
<tr>
<td>Modified actuator commands</td>
<td>Physical</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

### 2.4.2 Countermeasures

This work proposes a component-level security scheme (TECEP) that works to stabilize the plant in the event of an attack. Fig. 2.5 demonstrates that TECEP components create a last line of defense against the threat vectors described in Section 2.4.1. These trusted components are isolated from attack and capable of intervening at the chip pin level. Sensor data can be compared against expected values to detect interruption or falsification. Malicious updates to the control code can be exposed by speculating about the future behavior of the controller and plant. The architecture of TECEP is described in detail in Chapter 4.

Similar runtime monitoring and fault-detection methods have been proposed by Sha [23] and Dai et al. [24]. Their proposed mechanisms monitor new sensor measurements and the
controller’s reaction to them. TECEP differentiates itself in its efforts to be transparent to a control system designer. Moreover, supervisor and monitoring functions are implemented in hardware to reduce vulnerability to malicious software attacks. Most importantly, these previously proposed systems are intrinsically reactive. The plant comes under duress and corrective action must be taken. TECEP proposes a means of predicting and preempting attacks before they can negatively affect the plant. This prediction and preemption is accomplished by secure autonomic hardware and software.

Table 2.3 explores the options for separating a SoC into a Normal Zone (NZ) and a Secure Zone (SZ). A conventional single software kernel has no such separation. Even the kernel of a real-time OS is far too complex for the current capabilities of formal analysis tools. Without verification, some exploitable bugs may remain in the kernel. The only provisions for secure execution or data are software privileges and perimeter security. As discussed earlier, these mechanisms have proven insufficient.

Trusted execution environments like ARM’s TrustZone partition a system into a NZ and SZ [25]. In a TrustZone-enabled processor, some hardware resources like memory and interrupt controllers are allocated for SZ access only. SZ software may consist of anything from a library to an operating system. SZ software is isolated from NZ software, but some hardware
resources like the processor are shared between kernels. Even with the potentially reduced complexity of a SZ kernel, it still exceeds the current capabilities of formal analysis tools. SZ kernels are not immune to the same bugs that plague conventional kernels [26]. Although hardware is shared, some bus peripherals like memory and I/O controller are aware of which kernel is currently running, and the SZ kernel can utilize a secure boot procedure.

TECEP is designed such that it does not suffer from the above vulnerabilities. Both hardware and software components of the SZ are completely isolated from those of the NZ. Critical components are simple enough to allow the use of formal analysis tools. After verification, these components cannot be updated without physical access to the platform.

2.4.3 Assumptions and Limitations

TECEP, as described in this work, may not be directly applicable to legacy or proprietary systems, wherein a model of the plant is not readily available. Because a model is integral to TECEP and particularly to prediction, these systems would forgo a prediction unit and require a more heuristic approach. For example, in an automotive application some variation of TECEP could intervene to stop the car if sensors indicate a collision is imminent.

The threats described in Table 2.2 which require physical access are not emphasized in
this work. However, TECEP may be able to detect these threats. Table 2.4 shows the capability of TECEP to address internal and external threats. Latent malware within the control code can be detected via speculative execution of a virtual control loop. If sensors indicate normal plant operation when the model does not, it may indicate sensor tampering. Conversely, if sensors report an endangered plant when the model suggests normal operation, control commands may have been intercepted at the actuator. In the latter cases, TECEP is unable to take corrective action, but may sound an alarm or invoke a failsafe mechanism.

This detection of physical attacks is not infallible. The plant is expected to deviate from the model, and the magnitude and rate of acceptable deviation must be determined before deployment. Furthermore, if both the actuator and sensors are compromised, the PLC and TECEP are working blindly, with potentially no indication of error.

<table>
<thead>
<tr>
<th>Threat</th>
<th>Detection Method</th>
<th>Addressable by TECEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious control code</td>
<td>Speculative plant output</td>
<td>Yes</td>
</tr>
<tr>
<td>Sensor interference</td>
<td>Plant/model inconsistency</td>
<td>Partially</td>
</tr>
<tr>
<td>Modified actuator commands</td>
<td>Plant/model inconsistency</td>
<td>Partially</td>
</tr>
</tbody>
</table>

The fundamentally trusted TECEP components are the specification guards (Section 4.2.2) and control logic (Section 4.2.3), which monitor the plant’s behavior and have the final authority over commands sent to the plant. These components are defined in C/C++ and can be verified using open-source formal analysis tools. The details of verification are beyond the scope of this work. All other TECEP additions are trusted, though this assumption inherently requires trust in both control system designers and their design tools. Similarly, implementation tools like high-level synthesis and compilers are trusted as well. This thesis does not directly address verification and trust assurance, but instead focuses on the implementation of TECEP on a specific platform using high-level synthesis.
Chapter 3

Programmable System-on-Chip

Overview

The TECEP architecture, which includes various components that must be kept distinct, lends itself well to a system-on-chip (SoC) implementation. Because of the symmetry of hardware and software components as well as the required flexibility in the division of hardware and software, a programmable SoC is even more appropriate.

A system-on-chip assimilates a microprocessor, volatile memory, peripheral I/O controllers and assorted specialized hardware elements. Compare this with a microcontroller, which also incorporates a microprocessor, I/O controllers and a more limited amount of volatile memory, and is traditionally targeted at embedded applications. A programmable SoC follows the composition of a traditional, but includes programmable hardware fabric in lieu of specialized hardware modules. This work specifically targets Xilinx’s Zynq-7000 All Programmable SoC [27].
3.1 Target Platform

Xilinx’s Zynq-7000 SoC is divided into two sectors: the processing system and the programmable logic. The processing system (detailed in Section 3.1.1) can be viewed as a microcontroller featuring an ARM Cortex-A9 embedded processor. The programmable logic is equivalent to a low-cost field programmable gate array (FPGA), which contains programmable hardware and is detailed in Section 3.1.2. A block diagram of the Zynq-7000 can be seen in Fig. 3.1. Experiments for this work were conducted on an Avnet ZedBoard [28], a development board which contains a Zynq-7020 and sundry memories and peripherals.

3.1.1 Zynq Processing System

The processing system in the Zynq-7000 consists of an application processor unit (APU), memory controllers and peripheral controllers. The APU is based on an ARM Cortex-A9 dual-core embedded processor. Although the fastest speed grade is capable of clock speed up to 1GHz, for this work the ARM Cortex-A9 operated at its default frequency of 667 MHz. Each processing core features its own vector and floating point unit, a memory management unit (MMU) and 64KB of Level 1 (L1) cache memory (32KB instruction, 32KB data). 512KB of L2 cache is shared between the cores, along 256KB of on-chip SRAM. The APU also includes an eight-channel DMA (direct memory access) controller, system control registers, a global interrupt controller and various timers.

Outside the APU, the Zynq processing system contains myriad interconnects, interfaces and peripheral controllers. An external memory interface allows both the processing system and the programmable logic to be connected to external DDR memory. In the case of the ZedBoard, 512MB of DDR3 memory is available. Interfaces to NOR, NAND, and QSPI external flash memory are included for nonvolatile storage options.

Table 3.1 shows the peripheral controllers available on the Zynq. External storage is available via USB flash drives, SD cards or SPI-connected flash. Gigabit Ethernet provides high speed
Table 3.1: Zynq I/O controllers

<table>
<thead>
<tr>
<th>I/O Type</th>
<th>Number Available</th>
<th>Example Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD flash</td>
<td>2</td>
<td>External storage</td>
</tr>
<tr>
<td>General Purpose I/O</td>
<td>1</td>
<td>LED, switch, etc</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>2</td>
<td>Network access, high-speed communication</td>
</tr>
<tr>
<td>UART</td>
<td>2</td>
<td>Communication</td>
</tr>
<tr>
<td>CAN</td>
<td>2</td>
<td>Communication</td>
</tr>
<tr>
<td>USB</td>
<td>2</td>
<td>External memory, communication</td>
</tr>
<tr>
<td>SPI</td>
<td>2</td>
<td>External memory, low-speed communication</td>
</tr>
<tr>
<td>I2C</td>
<td>2</td>
<td>External memory, low-speed communication</td>
</tr>
</tbody>
</table>

communication. Lower-speed communication is available via USB, UART, CAN, SPI, and I2C. A GPIO module allows control over LEDs, switches, and various I/O pins. These I/O modules are memory-mapped and multiplexed so as to conserve pins when certain modules are unused. Multiple general-purpose and high-performance interconnects allow interfacing of the processing system with the programmable logic. An accelerator coherency port (ACP) provides direct access to L2 cache and on-chip memory for modules in the programmable logic. A cryptography block and an analog to digital converter (ADC) are located within the programmable logic but are available for the processing system’s use.

3.1.2 Programmable Logic

Depending on the package chosen, Zynq’s programmable logic is equivalent to either a low-cost (Xilinx Artix-7 [29]) or high-performance (Xilinx Kintex-7 [30]) field programmable gate array (FPGA). An FPGA is a semiconductor device that has no predetermined function. Unlike an application-specific integrated circuit (ASIC), whose operation is fixed during the manufacturing process, an FPGA’s function can be programmed repeatedly after manufacturing. FPGAs are an ideal platform for prototyping potential designs or for distributing a
Typically, a function is implemented in hardware in order to improve performance or power consumption over a similar software implementation. Because an FPGA is a “blank slate” of programmable hardware, a function can be duplicated numerous times with each instance running in parallel, thus improving performance. While not commonly considered low-power devices, an FPGA implementation of a highly-parallel function may draw considerably less power than the same operation implemented on a PC. Performance and power are the conventional reasons for FPGA usage, but this work explores the security benefits of using FPGA hardware.

A generic means of creating combinational hardware is by using lookup tables (LUT). LUTs simply store the appropriate output corresponding to each potential input of a function. Flip-flops (FF) are available at the outputs of LUTs as a storage element. LUTs and FFs are grouped into slices, and slices are combined into configurable logic blocks (CLB).

A typical FPGA architecture is shown in Fig 3.2, sans some resources like clock managers. A sizable number of CLBs are included, along with block RAM (BRAM) for use when a large amount of memory is required. Digital Signal Processing (DSP) slices allow for faster processing of common arithmetic operations. I/O blocks (IOBs) interface the design with external pins. These components connect via a programmable interconnect. Connections may usually be made at the intersection of any two routing paths, permitting cross-chip connections and communication.

Fig. 3.3 features a closer view of the CLBs, BRAMs, and DSP slices in the Zynq-7000. Each CLB is comprised of two slices, for a total of four LUTs and eight FFs. LUTs can be configured as 6-input/1-output or 5-input/2-output. BRAMs are dual-port and contains 36Kb of memory; each can addressed as a single unit or as two independent 18Kb BRAMs. DSP slices include a pre-adder, adder, 25x18 multiplier and an accumulator. Table 3.2 lists the available resources in the Zynq-7020 used for this work.
3.2 Programmable System-on-Chip Design

Embedded and SoC design traditionally involve hardware/software co-design, i.e., disparate but complementary designs of software and hardware [31]. A broadly-accepted, all-encompassing standard or language for system design has yet to arrive. The design process therefore typically begins with the partitioning of the system operations into hardware and software, usually in non-formal language. Each half of the system design is then captured in its respective language: a hardware description language (HDL) and a high-level software language (C, C++, etc). The two partitions then undergo simulation, verification and implementation, and are only recombined at the final stage before deployment.

The design flow of a programmable system-on-chip (PSoC) is shown in Fig. 3.6. A model-
based design may be used, which produces C/C++ code. The key feature of this design flow is that the C/C++ captured design may be used for both the hardware and software branches through the use of high-level synthesis (HLS). Simulation and verification may then be unified, and implementation follows well-established paths. HLS does have some shortcomings described in Section 3.2.2. However, for the majority of embedded and SoC designs, it is able to simplify and accelerate the design process.

3.2.1 Model-Based Design

The target application for this work is an ICS, discussed in Chapter 2. ICS development typically follows a model-based design flow [32][33], as shown in Fig. 3.7. System specifications and architecture are detailed, such as expected plant behavior, stability limits and
access policies [34]. Once the specifications have been established, a model of the plant is created using a software suite such as MATLAB/Simulink [35]. An existing physical plant may be observed and a mathematical data formed from the collected data. Alternatively, well-established equations representing mechanical, electrical, thermal and other physical quantities may be used create a model. A controller is developed based on the model plant behavior. The controller and plant model are simulated, and the controller is modified until the plant model responds to its inputs as expected. Physical plant hardware may then be used to further verify the controller via hardware-in-the-loop (HIL) simulation. HIL simulation is useful for eliminating real-world errors that may not be seen in a model plant. Finally, when the expected plant outputs are achieved, the system is deployed via C/C++ code produced by the modeling tools.
Table 3.2: Zynq-7020 resources

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Number Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup Table</td>
<td>53,200</td>
</tr>
<tr>
<td>Flip-flop</td>
<td>106,400</td>
</tr>
<tr>
<td>Block RAM</td>
<td>140</td>
</tr>
<tr>
<td>DSP Slice</td>
<td>220</td>
</tr>
<tr>
<td>I/O Block</td>
<td>200</td>
</tr>
</tbody>
</table>

### 3.2.2 Hardware Development

Hardware design traditionally involves RTL (register transfer level) modeling through the use of an HDL. However, using HLS technology, a high-level language such as C/C++ can be synthesized directly to RTL. HLS permits the convenience and simplicity of capturing a design at a high level of abstraction without the need for manual translation from C/C++ to HDL.

HLS does have some shortcomings which limit its usefulness [36]. Primary restrictions include the use of dynamic memory allocation and recursive functions. Dynamic memory allocation is unsupported because once the design has been specified, further resources cannot be allocated. The absence of a stack structure restricts the use of recursion. Arrays are favored over pointers when reading from memory. Because of the higher abstraction level, fine control over variables, inputs and outputs is also limited.

Once the HLS tools produce RTL, the hardware design follows a traditional flow. RTL is synthesized into a list of modules and their connections called a netlist. This netlist is passed to implementation tools, which map, place and route the design into the programmable fabric. Because it is a constrained optimization process, implementation is performed iteratively. The tools first attempt to maximize performance and minimize resource usage, modifying placements and routing as interconnect conflicts arise. After the design has been successfully mapped to the programmable logic, a stream of configuration bits (bitstream) is produced.
3.2.3 Software Development

Software design in a PSoC follows a familiar flow. The C/C++ code generated by the modeling tools is translated by the compiler into processors-specific object files. In the case of the ARM Cortex-A9, the ARMv7 instruction set architecture is used [37]. A linker/loader then combines these object files and any associated files into a single and executable file in the executable and linkable (ELF) format. This ELF file is then combined with the programmable hardware’s bitstream and downloaded to the target platform.
3.3 TECEP on a Programmable SoC

Because of the diverse set of components and a need for separation, TECEP lends itself well to a PSoC architecture. PSoC design follows traditional branches of hardware and software design, but utilizes a common source for both branches. The Zynq-7000 SoC is targeted, with software components running on its embedded ARM Cortex-A9 processor and hardware components implemented in its programmable fabric. Chapter 4 describes the components used in TECEP and Chapter 5 details their mapping to hardware and software.
Figure 3.7: Model-based design flow
Chapter 4

TECEP Security System Overview

Fig. 4.1 shows the logical components of the system. The darker components are those traditionally found in an industrial control loop. The remaining components are the result of integrating TECEP into the ICS design. Included are a production controller, backup controller, plant model, prediction unit and control logic and specification guards. The primary feature of this architecture is the separation of components—also denoted by the coloring used in Fig. 4.1 and detailed later in this chapter.

Both the production and backup controllers operate on the output of the physical plant. Predefined control logic and specification guards determine which controller’s output will be passed to the physical plant. The prediction unit, consisting of a copy of the production controller and a model of the plant, is used to preview any latent malware that may exist in the production controller. An additional model of the plant may be used to test sensor data input to the physical plant or to monitor the physical plant’s deviation from its ideal behavior.
4.1 Production Controller

The production controller operates just as it would in a traditional ICS, and can be implemented in software functions running on an embedded processor. An I/O module receives input from the plant, and passes it to the controller. The controller processes this input and produces an output, which is passed to the physical plant through another I/O module.

Regular updates to this controller can be made available via a network connection. Because it is exposed to a network, absolutely no trust is placed in the production controller. It therefore has no knowledge of the functions or existence of the added TECEP components. More specifically, it is unaware of the testing of its output by the specification guards before the output reaches the physical plant. This separation is crucial to security; its implementation...
will be detailed in Chapter 5.

4.2 System Monitor

![Diagram 1](A. Unprotected ICS)

![Diagram 2](B. ICS with TECEP Additions)

Figure 4.2: TECEP component isolation

The backup controller, specification guards, control logic and plant model are collectively referred to as the *system monitor*. These components are the most trusted in the system, and therefore have the highest authority. The system monitor should have no network access whatsoever, and updates should require physical access to the system. Fig. 4.2 shows the insertion of the system monitor and prediction unit into a traditional ICS. Any malware in the production controller will likely assume its outputs are sent directly to the plant. If it has any awareness of the TECEP components, they appear only as a “black box”, with no
means of internal access.

4.2.1 Backup Controller

The backup controller is an additional protected controller that can act upon the plant in the event that the production controller fails or is compromised by malware. Ideally, the backup controller is a legacy version of the production controller that has been simplified and optimized for stability. However, it may be an identical copy of the production controller that has been thoroughly verified and proven to be lacking any malicious or faulty behavior. The backup controller’s integrity can be verified and trusted because it is sequestered from the network. Updates would therefore require physical access to the system.

Commensurate with the production controller, the backup controller receives input from the physical plant, processes it, and produces an output to be returned to the plant. So long as the production controller’s output is within an acceptable range, the backup controller’s outputs are ignored by the system. However, if the production controller fails to produce an acceptable output, the backup controller supersedes it and enables the plant’s continued operation. The backup controller may also take over operations if the production controller fails to return any output within an allocated time.

4.2.2 Specification Guards

Specification guards are needed to determine the acceptable limits of both the system and of the inputs and outputs of individual components. ICSes are an ideal target for TECEP as these specifications already exist in the ICS design process. The details of the specification guards are unique to each system. As a baseline, all systems should include a fixed range of values that the plant’s output is expected to remain within, as well as the expected response times of individual components.
4.2.3 Control Logic

The control logic is a compact set of statements that checks the outputs of the physical plant, plant model and prediction unit against the system’s specification guards. If any of these values is not within the acceptable range, the production controller’s output is ignored and the backup controller’s output is passed to the physical plant. Depending on the application, this transition may be unidirectional, requiring a hard reset to return to the production controller. If productivity is a priority and stability is not absolutely critical, the control logic may allow for a return to the production controller without a reset.

Such a reverse may be useful in the event of a denial-of-service (DoS) attack. However, care should be taken to ensure that a fault is indeed temporary and that the production controller is not reinstated prematurely. A sample precaution would be to institute a “probation” period after a fault, during which the backup controller retains control of the plant regardless of the production controller’s value. If the production controller’s output is acceptable throughout its probation, it may be reinstated afterward.

Some leniency in the control logic may be useful to avoid unnecessary switching in the event of a brief disturbance. One possible implementation would be a counter connected to the specification guards to track the number of faults. In order for a switchover to occur, the number of faults within a given time period must exceed some predefined value. The addition of a Kalman filter would substantially reduce noise in the system and with it the likelihood of false error reporting. As with the specification guards, the conditions under which a switchover in either direction should occur is application-specific and therefore unspecified.

4.2.4 Plant Model

A model of the plant is included in the system monitor. The plant model is a preexisting resource created in the ICS design process. Although normally discarded once the design phase is complete, in this work it is preserved and included in the runtime implementation.
The plant model receives the output of the production controller. Its behavior is compared against that of the physical plant; the exact semantics of this comparison is application-specific. It may be used to detect attacks on sensor data: if the controller applies the same output to both the physical plant and the model, but the physical plant does not respond as expected, it may indicate an attack on the plant’s sensors. The comparison between the physical plant and the model may also be used to detect critical flaws by measuring the rate and/or magnitude of the physical plant’s divergence from the model. Periodic synchronization of the plant model with the physical plant may be necessary.

### 4.3 Prediction Unit

![Prediction Unit Diagram](image)

Although the system monitor will stabilize the plant when malicious behavior is detected,
that stabilization is reactive. The addition of the prediction unit allows the system monitor
to detect any latent malware embedded in the production controller and preempt its attempt
to destabilize or destroy the plant. Using the predictive information, the system monitor
can move the plant to a stable state before any malicious output of the production controller
reaches it.

The prediction unit is shown in Fig. 4.3. It consists primarily of a copy of the production
controller and a model of the plant. In order to mimic the real production controller as
closely as possible while maintaining security, the prediction unit’s controller is implemented
on an embedded processor as well. For the sake of both performance and simplicity, its
associated plant model is implemented on the same processor.

A supervisory function manages the prediction unit’s mode of operation. Four modes are
available: Normal, Accelerate, Save and Restore. When in Normal mode, the prediction unit
runs for one cycle as a simple closed loop of controller and plant. Save mode stores the current
state of the controller and plant. In Accelerate mode, the closed loop runs continuously
for a predefined number of cycles in order to approximate the future behavior of the real
production controller and physical plant. Restore mode restores the state information of the
controller and plant to the previously saved values.

During each system cycle, the prediction unit is run in its four modes. It is first run in
Normal mode for one cycle in order to duplicate the current behavior of the real production
controller. The respective states of the controller and plant are then preserved in Save mode
before acceleration. Accelerate mode runs for some user-defined number of cycles into the
future, and the future state of the plant is passed to the I/O function for output. Both
the controller and plant are then restored to their previous states before acceleration using
Restore mode.
4.4 Junction Box

All components in the system are connected via a junction box. Fig. 4.4 shows the connections between components in the system. The interface to the production controller includes its output \( (u_{\text{production}}) \) and an input from the physical plant’s \( (y_{\text{physical}}) \). The plant’s input \( (u_{\text{plant}}) \) and output \( (y_{\text{physical}}) \) are passed through the junction box directly so as to minimize delays. The prediction unit has a single output from its accelerated plant \( (y_{\text{accelerated}}) \). The system monitor has the largest interface with the junction box, including the production controller’s output \( (u_{\text{production}}) \), the physical plant’s output \( (y_{\text{physical}}) \), the prediction units output \( (y_{\text{accelerated}}) \), and status code originating in the junction box. The junction box manages any handshaking with the prediction unit and the system monitor.

![Diagram of Junction Box Component Connections](image)

Figure 4.4: Junction box component connections

Along with passing signals, the junction box manages the system’s flow of control, as shown in Fig. 4.5. The system cycle begins with the production controller. Once it produces an output, the prediction unit is set to begin operation. The junction box contains internal
Figure 4.5: Junction box flow of control
timers to monitor the response time of the production controller and prediction unit. If either fails to respond within its allotted time, a status code is updated with the relevant information. Regardless of the success or failure of these units, the junction box next passes all necessary values to the system monitor and signals it to begin operation. The controller output chosen by the system monitor is then passed to the plant, and the status of the system is assessed. If no error has occurred, the junction box waits for the next output from the production controller. If an error has been recorded, the junction box continues to operate the system monitor based on its own internal timer. Once a component has failed, a system reset is necessary to clear the status code and reinstate the production controller.

Despite its relative simplicity, the junction box should pass rigorous testing and verification. It is a critical piece of the system, connecting all components and managing the system’s flow of control. Like the system monitor, the junction box should be transparent to the production controller.

4.5 Component Trust and Separation

Fig. 4.6 modifies Fig. 4.1 to show the separation of trusted and untrusted components. The production controller is network accessible and updatable and therefore completely untrusted. Any functions sharing resources with the production controller are also untrusted. All TECEP components (backup controller, control logic, specification guards, plant model) are verified, trusted and invisible to the production controller.

The copy of the production controller within the prediction unit is untrusted as well. Although the plant model used in the prediction unit is identical to the one in the system monitor, it is considered suspicious as it does share resources with the production controller, as detailed in the Section 5.4. The physical plant is treated as a black box, and is assumed to be physically secured.

It is vital that untrusted components not have access to or awareness of trusted TECEP
components. This separation suggests the use of both software and programmable hardware. The details of component segregation are provided in Chapter 5.

Figure 4.6: Trust in system components
Chapter 5

TECEP Security System
Implementation

The security system described in Chapter 4 is implemented on the Zynq-7020 PSoC according to Fig. 5.1. The production controller is implemented on the embedded processor in the Zynq processing system. All other components are realized in the programmable logic. The prediction unit runs on a soft processor in the programmable logic. The junction box is created using HDL. All other components are implemented via HLS, and a system cycle time of 1 ms is used.

5.1 Production Controller – Software Design

In most applications, the production controller code is generated by model-based design tools like MATLAB/Simulink. For the example used here, a proportional-integral-derivative (PID) controller is derived from XAPP1163 [38]. Controller coefficients are listed in Table 5.1. For testing, the response to a uni step is used. In order to simulate a software reconfiguration attack, latent malware was inserted into the production control code. The malware lies
dormant for 350 system cycles, then attempts to drive the plant’s output to its maximum (clipped) value. To demonstrate the system monitor’s ability to reinstate the production controller in the event of a DoS attack, the malicious behavior ends after 300 system cycles.

The production controller is implemented in software functions on the Zynq’s ARM Cortex-A9. An I/O function interacts with the programmable fabric via an AXI bus. The software timing is based on an AXI Timer/Counter module in the programmable logic. At the start of each 1 ms cycle, the production controller receives the plant’s $y_{\text{physical}}$ output from its I/O function. It processes this input, and produces a $u_{\text{production}}$ output to be sent to the programmable fabric. After sending an output, the production controller waits for the next system cycle to begin.

For the purpose of this work, the motor (plant) was emulated in software function running on the ARM Cortex-A9. Whether real or emulated, each cycle begins with the plant receiving input and producing its output.
In its current state, these functions run as a “bare-metal” application, with no kernel or OS. As a next step, the controller and I/O will become tasks of a real time operating system (RTOS). If further functionality is necessary, such as a file system or network access, a more complex OS such as Linux may be used. Regular software updates can be made to the controller code via network-accessible, external flash memory. Because of its network access, all software (including controller code, supervisory functions, operating system and drivers) is untrusted. These modules are therefore isolated from and unaware of any TECEP components.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>-0.7931</td>
<td>Derivative constant</td>
</tr>
<tr>
<td>Gd</td>
<td>6.0324</td>
<td>Derivative gain</td>
</tr>
<tr>
<td>Gp</td>
<td>35.3675</td>
<td>Proportional gain</td>
</tr>
<tr>
<td>Gi</td>
<td>0.5112</td>
<td>Integration gain</td>
</tr>
<tr>
<td>clip_min</td>
<td>-64.0</td>
<td>Minimum controller output</td>
</tr>
<tr>
<td>clip_max</td>
<td>63.0</td>
<td>Maximum controller output</td>
</tr>
<tr>
<td>y_min</td>
<td>-3.2</td>
<td>Minimum sensor input from a stable plant</td>
</tr>
<tr>
<td>y_max</td>
<td>3.2</td>
<td>Maximum sensor input from a stable plant</td>
</tr>
<tr>
<td>w_safe</td>
<td>0.0</td>
<td>Quiescent reference controller input</td>
</tr>
<tr>
<td>w_step</td>
<td>1.0</td>
<td>Controller step-response input</td>
</tr>
</tbody>
</table>

5.2 System Monitor High-Level Synthesis

The system monitor, comprised of the backup controller, plant model, specification guards and control logic, is implemented in the Zynq programmable logic using Vivado HLS [39]. In keeping with the PID nature of the production controller, the backup controller is proportional-only, and uses the same proportional gain coefficient listed in Table 5.1. As mentioned in
Section 4.2.3, a probationary period has been built into the system monitor. If any unit is outside specification, the backup controller is invoked and a counter is set to some value. This counter decrements each cycle, and is reset if another fault (current or future) is detected. If the counter reaches zero, the production controller is eligible to be reinstated.

HLS allows the backup controller and plant model developed and verified in the model-based design process to be translated directly to programmable hardware. Because the specification guards and control logic are captured in C/C++, formal software verification can be applied to ensure correct operation [40]. To begin the HLS process, the software functions representing the system monitor components are collected into a .c file. A top level function is chosen (system_monitor) and C synthesis executed. An initial timing and resource estimation report is generated by Vivado HLS.

Design optimization takes place at this point in the process. A variety of directives are available for optimization, including loop unrolling, pipelining, resource restriction and array partitioning [36]. TECEP components are being placed in hardware in order to reduce malware vulnerability. Therefore resource minimization takes precedence over performance. The setdirectiveallocation command is used to restrict the number of instantiated floating point cores to one addition/subtraction unit, one multiplication unit and one comparator unit. A configbind command further restricts the resources consumed by each of the floating point cores. No performance-optimizing directives are used as the functions used are relatively simple. Moreover, resource conservation and performance enhancement are usually mutually exclusive goals.

Once optimization is completed, C synthesis is repeated to confirm reduced resource usage or improved performance. Interface types are chosen for each input and output unless the default configurations are acceptable. A list of all interface protocols and their compatibility with various data types is available in [36].

Table 5.2 shows the I/O of the system monitor. Clock and reset ports are created regardless of the selected interface. The input signals u_sw, y_physical and y_accel use the ap_none
protocol. This protocol uses no handshaking signals, creating only simple data ports. A `ap_ctrl_hs` protocol was applied to the function itself. This protocol creates handshaking signals for the synthesized hardware block, as well as a port for the function’s return value. `ap_start` signals the block to begin operation; `ap_done` indicates its operation is complete. When `ap_ready` is high, the block is waiting for new inputs. In a non-pipelined design, `ap_ready` is functionally equivalent to `ap_done`. `ap_idle` indicates the block has completed operation and awaits a new start signal. `ap_return` is the synthesized function’s return value. In the case of the system monitor, the return value is the selected controller output. The value on this port is valid when `ap_done` is asserted.

<table>
<thead>
<tr>
<th>I/O Name</th>
<th>Interface Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_sw</td>
<td>ap_none</td>
<td>Output of production controller</td>
</tr>
<tr>
<td>y_physical</td>
<td>ap_none</td>
<td>Output of physical plant</td>
</tr>
<tr>
<td>y_accel</td>
<td>ap_none</td>
<td>Output of prediction unit</td>
</tr>
<tr>
<td>ap_clk</td>
<td>ap_ctrl_hs</td>
<td>Clock</td>
</tr>
<tr>
<td>ap_rst</td>
<td>ap_ctrl_hs</td>
<td>Reset</td>
</tr>
<tr>
<td>ap_return</td>
<td>ap_ctrl_hs</td>
<td>Output of system monitor</td>
</tr>
<tr>
<td>ap_start</td>
<td>ap_ctrl_hs</td>
<td>Begin operation</td>
</tr>
<tr>
<td>ap_done</td>
<td>ap_ctrl_hs</td>
<td>Operation is complete</td>
</tr>
<tr>
<td>ap_ready</td>
<td>ap_ctrl_hs</td>
<td>Ready for new inputs</td>
</tr>
<tr>
<td>ap_idle</td>
<td>ap_ctrl_hs</td>
<td>Waiting for start signal</td>
</tr>
</tbody>
</table>

5.2.1 Interface Synthesis

Interface adapters may be added during synthesis if the synthesized block requires an AXI protocol. An AXI4-Master, AXI4-Lite Slave or AXI4-Stream adapter may be added to supported interface types, as described in [36]. These adapters allow the block to be connected to the ARM Cortex-A9 processor, a DMA engine, BRAM, etc. During the testing phase, an
AXI4-Lite Slave adapter was added to the system monitor. A software testbench was then able to access to its return value and handshaking signals via memory-mapped I/O. In the final design, however, the system monitor uses no interface adapters.

After C and interface synthesis complete, the design is exported as either a Pcore to be used in Xilinx ISE or as an IP block for use in Xilinx’s Vivado Design Suite (Vivado). The export process creates RTL files from the provided C code, which include all I/O and interface adapters. Preliminary versions of the system monitor are exported as Pcores, but the final design is exported as IP to Vivado.

Interface synthesis presents a simple means of attaching an AXI interface to an existing IP. Vivado requires the user to create a custom IP and delve into the AXI interface’s HDL. In order to expedite the process of connecting the junction box to the ARM Cortex-A9 and MicroBlaze, three interface IPs were created in Vivado HLS.

Identity C functions, which simply return their input argument, were used to create the adapters. For the production controller and prediction unit, the `ap_hs` protocol is chosen in order to perform blocking reads on their outputs. An `ap_none` protocol is used for the emulated physical plant to minimize the impact on its I/O. With a real physical plant, this adapter is not necessary. The three interface IPs are synthesized and exported to Vivado’s IP catalog. Fig 5.2 shows their connections between the junction box, ARM Cortex-A9 and MicroBlaze.

![Figure 5.2: HLS interface adapters](image-url)
5.3 Junction Box – Hardware Design

The junction box is captured in Verilog HDL using Vivado, and verified with Vivado’s built-in simulation tools. Watchdog timers are incorporated to monitor the response time of the production controller and the plant model. Fig. 5.3 shows the internal control logic of the junction box. If either of the production controller or prediction unit watchdog timers expires, the status is updated for the system monitor and the junction box continues operation based on its own internal 1 ms timer.

![Figure 5.3: Junction box internal control](image)

5.4 Prediction Unit – Soft Processor

The prediction unit is implemented in software functions on a MicroBlaze soft processor core [41]. Using a soft processor allows close mirroring of the production controller’s implementation while still maintaining the segregation of all TECEP components. An identical copy of the production controller code runs on the MicroBlaze, along with a copy of the plant model included in the system monitor and an I/O function. An AXI Timer/Counter is
included to measure prediction unit runtimes. As on the ARM Cortex-A9, the MicroBlaze runs a bare-metal application with no OS or kernel. Instead, a supervisory function was written in C to regulate the prediction unit.

Table 5.3 shows the four modes of operation available in the prediction unit. During each system cycle, the prediction unit iterates through these steps once it receives a start signal from the junction box. First, the virtual closed loop is run in Normal mode for one cycle in order to synchronize with the production controller and physical plant. Save mode then stores the current values of the state variables of both the controller and the plant. Accelerate mode runs the closed loop for some number of cycles in order to approximate the future behavior of the physical plant. After acceleration, the accelerated plant’s output is passed to the system monitor. Restore mode reinstates the plant and controller state variables to their values before acceleration, and the prediction unit waits for the next start signal.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save</td>
<td>Save the current states of the controller and plant</td>
</tr>
<tr>
<td>Restore</td>
<td>Restore the controller and plant to their previously saved states</td>
</tr>
<tr>
<td>Normal</td>
<td>Run the closed loop for one cycle</td>
</tr>
<tr>
<td>Accelerate</td>
<td>Run the closed loop N cycles</td>
</tr>
</tbody>
</table>

5.5 Component Integration

The system was implemented using Vivado 2013.2 on a 64-bit Linux workstation host running the 2.6.32-28-generic kernel with an 4-core 2.8 GHz Intel Core i7 processor and 24GB of RAM. A new project was created in Vivado, along with a new block design. Fig. 5.4 shows the IP included in the block design. The ARM Cortex-A9 operates at its default frequency of 667 MHz and utilizes its hardware floating point unit (FPU). All components in the
programmable fabric operate at 140 MHz. This is the highest attainable frequency for the PL in this design; the limiting factor appears to be the MicroBlaze’s local memory. The MicroBlazewas configured with a hardware FPU, 5-stage pipeline and 16KB of local memory. Full (-O3) compiler optimization was used for both processors.

After all modules were added, design rule checks ensured that all blocks were correctly instantiated and no connections were missing. A netlist was then produced by synthesis and passed to the implementation tools. Implementation mapped the design to the hardware and produced a bitstream. The hardware configuration and bitstream were exported to Xilinx SDK. Within the SDK, the software described in Section 5.1 was created and compiled. The resulting ELF executable and the bitstream were then downloaded to the device.

Figure 5.4: Vivado block design
Chapter 6

Implementation Results

TECEP, the security architecture described in Chapters 4 and 5, was demonstrated using a motor control application based on XAPP1163 [38]. The motor itself was emulated in software, and all components were implemented on the Zynq-7000 SoC. Results are described in the follow sections.

Table 6.1: Motor Controller-specific attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probation period</td>
<td>200 system cycles</td>
<td>Number of cycles after a fault before production controller is eligible for reinstatement</td>
</tr>
<tr>
<td>Prediction window</td>
<td>200 system cycles</td>
<td>Number of cycles virtual control loop is run in Accelerate mode</td>
</tr>
<tr>
<td>Production controller</td>
<td>200 $\mu$s</td>
<td>Time allowed for production controller operation before it is considered unresponsive</td>
</tr>
<tr>
<td>watchdog timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction unit</td>
<td>750 $\mu$s</td>
<td>Time allowed for prediction unit operation before it is considered unresponsive</td>
</tr>
<tr>
<td>watchdog timer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.1 Application-specific Attributes

Some of the attributes described in Chapter 5 are specific to each platform and application. Table 6.1 lists the values specific to this motor control application.

The system monitor’s probationary period for the production controller was chosen to be 200 system cycles to match the size of the prediction window. A counter is set to a value of 200 each time a present or possible future fault is detected. The value of this counter can be thought of as the fourth input to the specification guards (Fig. 4.1).

The number of acceleration cycles depends on the application needs and the target hardware. All operations described in Fig. 4.5 must be completed within the system’s 1 ms cycle time. A 140 MHz clock was used for the Zynq PL in this design. Each Normal iteration requires roughly 1.43 $\mu$s for completion. Based on this runtime, along with the comparatively negligible runtimes of the production controller, system monitor and I/O modules, the prediction unit is capable of speculating 500 cycles into the future during each system cycle. For this motor control application, however, only a 200-cycle prediction window was used.

Because performance was not a priority, somewhat arbitrary timeout values of 200 $\mu$s and 750 $\mu$s were chosen for the production controller and prediction unit, respectively. Both units should complete their operations well within these limits. If timing is critical, these values should be more carefully chosen to reflect each component’s exact runtime.

6.2 Motor Controller with TECEP Additions

In order to test the effectiveness of TECEP, latent malware was inserted into the production control code. This malware attempts to drive the motor to its maximum output beginning at $t = 350$ ms. The various stages of TECEP integration are displayed in the following figures.

- Fig. 6.1 shows the output of the plant when no countermeasures are enabled. The
system begins with a unit response, followed by the malicious behavior. The controller requires roughly 130 ms to drive the plant to its maximum (clipped) output of 3.33. If a physical motor were in use, this stress would likely cause it to malfunction after some time.

- Fig. 6.2 shows the addition of the TECEP components. The infected controller again attempts to drive the plant out of bounds at $t = 350$ ms. Once the system monitor measures the plant output as greater than $y_{\text{max}}$ (3.2), it invokes the backup controller to return the plant to a known stable state ($w_{\text{safe}} = 0.0$) within 150 ms. Despite this recovery, the malware is successful in destabilizing the plant. Fig. 6.2 also includes the protection-less plant output for reference.

- Fig. 6.3 exhibits the full capabilities of TECEP including prediction. The system be-
gains with same unit response and malware. The plant becomes unstable at $t = 480$ ms. Using a prediction window of 200 system cycles, the backup controller can be preemptively invoked at $t = 280$ ms. The plant is stabilized before the malware can begin, and its effects never reach the plant.

- Fig. 6.4 features a more resilient design, which reinstates the production controller when its malicious behavior ends. The malware again starts at $t = 350$ ms, but here the malware is temporary and ends at $t = 650$ ms. Using the predicted plant output, the system monitor invokes the backup controller at $t = 280$ ms and starts the production controller’s probation counter. Continuing with prediction, the system monitor can see that the malicious commands end and that the production controller will attempt to return the plant to its acceptable range by $t = 655$ ms. Using its 200 ms
Figure 6.3: Plant output with TECEP additions and prediction

prediction window, the system monitor releases the probation counter at \( t = 455 \) ms. Since the probation period is also 200 ms, it expires at \( t = 655 \) ms. At that time, the production controller is reinstated and it returns the plant to full productivity. Although appropriate for this application, it is not necessary that the probation period and prediction windows be of equal length.

6.3 Resource Utilization

Resources consumed by the TECEP components are modest, allowing for a more compact programmable fabric or a more robust system monitor. The hardware monitor consumes 677 FFs, 1046 LUTs and 5 DSP slices. The DSPs are utilized in the floating point addition,
Figure 6.4: Plant output with return to production controller

multiplication and comparator cores. In total, the four HLS-generated interface adapters described in Section 5.2.1 require only 295 FFs and 78 LUTs. Because the junction box’s functionality consist mostly of passing signals, it is lightweight at 70 FFs and 80 LUTs. The prediction unit is the most expensive of the TECEP components in terms of resource utilization. Two DSP slices support the MicroBlaze’s mathematical operations; four BRAMs are required for its 16KB of local memory. The soft processor is constructed with 2813 FFs and 3174 LUTs.
Table 6.2: Programmable logic resource utilization

<table>
<thead>
<tr>
<th>Zynq-7020</th>
<th>FF</th>
<th>LUT</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Monitor</td>
<td>677</td>
<td>1046</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>HLS interfaces</td>
<td>295</td>
<td>78</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Junction Box</td>
<td>70</td>
<td>80</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prediction Unit</td>
<td>2813</td>
<td>3174</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total Used</strong></td>
<td>3855</td>
<td>4378</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td><strong>Available</strong></td>
<td>53200</td>
<td>106400</td>
<td>140</td>
<td>220</td>
</tr>
<tr>
<td><strong>Percent Used</strong></td>
<td>7%</td>
<td>4%</td>
<td>5%</td>
<td>2%</td>
</tr>
</tbody>
</table>

6.4 Timing Results

The production controller operates at 667 MHz on the ARM Cortex-A9. The TECEP components in the Zynq PL use a 140 MHz clock (7.143 ns clock period). The system cycle time is 1 ms. Table 6.3 lists the runtimes of each component. Runtimes were measured using AXI Timers in the PL. The production controller software consists primarily of simple multiplication and addition, and runs quickly on the robust ARM Cortex-A9. The system monitor’s runtime is slightly longer, as it includes the plant model’s operation. One Normal cycle of the prediction unit requires significantly more time. The prediction operation includes the production control code, plant model and supervisor (mode-switching) function. Furthermore, it runs on a modest soft processor with a clock speed that is 79% slower than that of the ARM Cortex-A9.

Table 6.3: Component runtimes

<table>
<thead>
<tr>
<th>Component</th>
<th>PL Clock Cycles</th>
<th>Time(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production Controller</td>
<td>25</td>
<td>179 ns</td>
</tr>
<tr>
<td>System Monitor</td>
<td>31</td>
<td>221 ns</td>
</tr>
<tr>
<td>Prediction Unit</td>
<td>202</td>
<td>1430 ns</td>
</tr>
</tbody>
</table>
Significant timing overhead is found in the AXI communications between the junction box and the two processors. The AXI4-Lite interface only supports one data word per transaction; each transaction requires 15-20 clock cycles (107-143 ns). Faster interfaces are available with larger word sizes or burst transactions, but are more costly in terms of hardware resources. The lightweight AXI4-Lite was chosen because resource utilization took precedence over performance for this work.

6.5 Evaluation of Results

This simple motor control application successfully demonstrates the effectiveness of the TECEP architecture. Despite latent malicious behavior embedded in the control code, the system monitor is capable of intervening at the lowest level and stabilizing the plant. Furthermore, speculative execution using a model of the plant allows preemption of malware, such that its effects never reach the plant. The TECEP additions incur a very modest increase in hardware resources. Performance is comparable to a traditional full-software implementation, and therefore sufficient for the objectives of this work.
Chapter 7

Conclusions

7.1 Motivation

Embedded platforms have made computing ubiquitous. As a result, cyberattacks have become commonplace as well. ICSes have been retroactively networked for monitoring and maintenance, but largely consist of legacy designs in which security was not considered.

Awareness and concern over potential ICS attacks are growing. Experts are discussing ICS vulnerabilities and calling for new security solutions [42][43]. Probes and attacks of ICS honeypots have become more frequent, focused, coordinated and widespread [44].

Although perimeter security and firewalls are the contemporary approach to ICS security, they have proven insufficient. Successful attacks can break through weak authentication or be embedded in communication protocols. The effects of attacks like Stuxnet and Aurora could be catastrophic, as ICSes are used in the processes of water purification, food manufacturing, power generation and oil refining.
7.2 Contributions

This work describes a component-level security architecture focused on trust enhancement of critical embedded processes. TECEP incorporates a secondary, high-assurance backup controller which can stabilize the plant if the production controller is compromised. Such a compromise is detected by predefined specification guards. A model of the plant is an existing resource retained after the ICS design process in order to vet plant sensor data. Control logic monitors the specification guards and deploys the backup controller if the production controller is deemed unsuitable. These four components form the system’s root of trust and are collectively referred to as the system monitor. Duplicates of the plant model and the production control code are implemented as a prediction unit. This virtual control loop is run at a higher frequency than the primary, offering an informed speculation of the future behavior of the plant.

The system monitor and the prediction unit are realized in the programmable fabric of a Zynq-7000 programmable system-on-chip. Nearly all TECEP components are captured in C/C++. The system monitor can therefore be verified using formal software analysis tools, and is isolated in programmable hardware using Vivado High-Level Synthesis. The prediction unit runs in software functions on a MicroBlaze soft processor. The addition of TECEP is unobtrusive to the traditional ICS design process, and incurs a minimal increase in hardware resources.

7.3 Future Work

An immediate extension to the work of this thesis is the addition of a real-time operating system, as would be found in most industrial control application. An ARM soft processor [45] would allow precompiled production control code to be executed in the prediction unit without the need for a binary translation. Additionally, a theory of robust switching
between controllers should be developed.

Countermeasures for interference with communication between trusted TECEP blocks and I/O pins should be investigated. If a simple serial interface is used, a custom high-assurance controller may be implemented. Unfortunately, the use of a more complex protocol such as USB would complicate the matter considerably. Communication interference with internal signals should be investigated as well. If the OS running on the ARM Cortex-A9 or soft processor is compromised, it could potentially overflow the AXI bus or reconfigure multiplexed I/O pins.

Similarly, if the OS supervising prediction is compromised, any malware may detect that it is running in a virtual control loop. If so, the prediction unit may be nullified by the supervisor always reporting that the future state of the plant is safe. TECEP would still be capable of stabilizing the plant, but preemption would be lost. The feasibility of this threat and potential countermeasures should be investigated.

Experimental results are needed to further verify the work presented here. The Xilinx Zynq-7000 All Programmable SoC/Analog Devices Intelligent Drives Kit provides a smooth transition from the emulated motor used here to a physical DC motor [46]. TECEP’s application to robotics can be explored as well, in the spirit of Asimov’s Three Laws of Robotics [47].

In some applications, TECEP may be adapted to serve as a supervisor or monitor only, with limited intervention. A model of the plant may not be available in legacy or proprietary systems, thus making prediction infeasible. The system monitor may then operate on a predefined set of parameters, such as not allowing a motor to run at full capacity for an extended period of time. Even in this limited capacity, TECEP’s low-level implementation may still allow it to be more effective than traditional perimeter-focused security.
Bibliography


[18] Thomas C. Reed. *At the Abyss: An Insider’s History of the Cold War*.


[47] Isaac Asimov. *I, Robot*. 