S3A: Secure System Simplex Architecture for Enhanced Security and Robustness of Cyber-Physical Systems *

Sibin Mohan
Information Trust Institute
University of Illinois
Urbana IL 61802
sibin@illinois.edu

Stanley Bak
Dept. of Computer Science
University of Illinois
Urbana IL 61802
sbak2@illinois.edu

Emiliano Betti
System Programming
Research Group
University of Rome
"Tor Vergata", Rome
betti@sprg.uniroma2.it

Heechul Yun
Dept. of Computer Science
University of Illinois
Urbana IL 61802
heechul@illinois.edu

Lui Sha
Dept. of Computer Science
University of Illinois
Urbana IL 61802
lrs@illinois.edu

Marco Caccamo
Dept. of Computer Science
University of Illinois
Urbana IL 61802
mcaccamo@illinois.edu

ABSTRACT

The recently discovered 'W32.Stuxnet' worm has drastically changed the perception that systems managing critical infrastructure are invulnerable to software security attacks. Here we present an architecture that enhances the security of safety-critical cyber-physical systems despite the presence of such malware. Our architecture uses the property that control systems have deterministic (real-time) execution behavior to detect an intrusion within $0.6\ \mu s$ *while still guaranteeing the safety of the plant. We also show that even if an attacker is successful (or gains access to the operating system's administrative privileges), the overall state of the physical system still remains* safe.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-Time Systems and Operating Systems; Cyber-Physical Systems; Safety-Critical Systems*; D.4.6 [**Operating Systems**]: Security and Protection—*Intrusion Detection*

Keywords

Real-Time Systems, Intrusion Detection, Cyber-Physical Systems, Stuxnet, Safety-Critical Systems, Secure Simplex, S3A.

1. INTRODUCTION

Many systems that have safety-critical requirements such as power plants, industry automation systems, automobiles, *etc.* can be classified as cyber-physical systems (CPS) – *i.e.* a tight combination of, and co-ordination between, computational and physical components. These systems (or parts of them) have stringent safety requirements and require deterministic operational guarantees (including real-time properties). Such systems have also traditionally been considered to be extremely secure since they *(a)* are typically not connected to the Internet; *(b)* use specialized protocols and proprietary interfaces ('security through obscurity') *(c)* are physically inaccessible to the outside world and *(d)* typically have their control code executing on custom hardware such as specialized processors or programmable logic controllers (PLCs). This misconception of ironclad security, however, has recently been exposed when the '*W32.Stuxnet*' worm [1] targeted and successfully infiltrated a Siemens WinCC/PCS7 control system [7]. Not only did it bypass all the security (digital/physical) techniques but it also reprogrammed the PLC that controlled the main system and *caused physical damage to the system*.

In this paper, we specifically address the problem of security for physical control systems with real-time requirements. Compared to general-purpose techniques, our work is different in that we focus on domain-specific characteristics of these systems and in particular, their *deterministic real-time nature*. We introduce a system architecture where an isolated and trusted hardware component is leveraged to enhance the security of the complete system. We present a novel intrusion detection mechanism that monitors context-specific side channels on the main CPU and in our initial prototype we use the *deterministic execution profile* of the system for this purpose[2].

Hence, we present the *Secure System Simplex Architecture* (S3A) to improve the security of cyber-physical systems that uses a combination of *(i)* knowledge of high-level control flow *(ii)* a *secure co-processor* implemented on an FPGA[3] *(iii)* deterministic execution time profiles and *(iv)* System Simplex [2, 21]. S3A detects intrusions that modify execution times by as low a value as $0.6\mu s$ on our test control system. With S3A, we expand the definition of 'correct system state' to include not just the physical state of the plant but also the *cyber state*, *i.e.* the state of the computer/PLC that executes the controller code. This type of security is hard for

*This work is supported in part by a grant from Rockwell Collins, by a grant from Lockheed Martin, by NSF CNS 06-49885 SGER, NSF CCR 03-25716 and by ONR N00014-05-0739. Opinions, findings, conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

HiCoNS'13, April 9–11, 2013, Philadelphia, Pennsylvania, USA.

[1] henceforth referred to as just 'Stuxnet'

[2] We elaborate on other potential side-channels in Sections 5.3 and 9.

[3] Can be a trusted processor/unwritable FPGA in the final implementation

an attacker to overcome by reverse engineering the code or the system especially since it involves *absolutely no changes to the source code/binary*. Even if an infection occurs and all of the security mechanisms are side-stepped (such as gaining access to the administrative privileges or the replication of our benevolent side channels), the trusted hardware component (secure co-processor) and the robust Simplex mechanism will still prevent the physical system from coming to harm, even from threats such as Stuxnet. Sections 4 and 5 present the details about our solution.

It is important to note that S3A is a *system-level solution that integrates multiple different solutions* to achieve security and safety in this domain. While we picked some mechanisms (execution time, Simplex, *etc.*), other concepts (Section 8) can be integrated to make the system that more secure and robust.

As the **main contribution** of this paper, we present the *Secure System Simplex Architecture* (S3A) where,

1. A **trusted hardware component** provides oversight over an untrusted real-time embedded control platform. The design provides a guarantee of plant safety in the event of successful infections. Even if an attacker gains administrative/root privileges she cannot inflict much harm since S3A ensures that the overall system (especially the physical plant) will not be damaged.

2. We investigate and use **context-dependent side channels for intrusion detection**, monitored by the trusted hardware component. They qualitatively increase the difficulty faced by potential attackers. Typically side-channel communication is used to break security techniques but we use them to our advantage in S3A. In this paper, we focus on side-channels in the context of CPU-controlled real-time embedded control systems as explained in Section 5.

3. We build and evaluate an **S3A prototype** for an inverted pendulum plant and discuss implementation efforts and the construction of side channel detection mechanism for *execution time-based side channels* using and FPGA in the role of the trusted hardware component. The side channel approach is shown to detect intrusions significantly faster than earlier plant-state-only detection approaches. This is explained in Section 5.4.

Further information on background, threat models, *etc.* is provided in Sections 2, 3 and Section 4

While intrusion detection is a broad area in computer security, our approach takes advantage of the real-time properties specific to embedded control systems. Also, most of the existing side-channel techniques/information (timing, memory, *etc.*) have traditionally been used to break the security of systems. This paper proposes a method so that these pieces of information are now used for *increasing* the security of the system. Also, such techniques have not been used before with the perspective of safety-critical control systems – hence we believe that this paper's contributions are novel.

We believe that our approach is generalizable to PLC and microcontroller-based CPS. Our justification is twofold; such systems *(i)* have stringent requirements for correct operation, *i.e.* the physical state of the plant must be kept safe under all conditions and *(ii)* often require the controller process to be deterministic.

Assumptions:

Important assumptions for the work presented in this paper are: *(a)* the system consists of a set of periodic, *real-time* tasks with stringent timing and deadline constraints managed by a real-time scheduler; such systems typically do not exhibit complex control flow, do not use dynamically allocated data structures, do not contain loops with unknown upper bounds, don't use function pointers, *etc.* – in fact, they are often designed/developed with simplicity and determinism in mind *(b)* the hardware component must be trusted and can only be accessed by authorized personnel/engineers – this is not unlike the RSA encryption mechanism where the person holding the private key must be trusted *(c)* while we use an FPGA for a prototype implementation, the final hardware component could be implemented on an ASIC or custom processor or even an FPGA with its programmability turned off to prevent further tampering *(d)* the systems we describe are rarely updated and definitely not in a remote fashion (unlike, say, mobile embedded devices)[4].

Note: Our techniques are not specific to attacks mentioned in this paper (especially those in Section 2) and tackles the broader class of security breaches of controllers in safety-critical CPS.

2. MOTIVATION

Many control systems attached to critical infrastructure have traditionally been assumed to be extremely secure. The chief concern in such systems is *safety*, *i.e.* to ensure that the plant's operations remain within a predefined safety envelope. "Security" was attained by restricting access to such systems – no connection to the Internet and only a few people could access the computers that controlled these systems. Also, parts (or even all) of the control code executed on dedicated hardware (PLCs for instance).

2.1 Stuxnet

The *W32.Stuxnet* worm attack [7] overturned all of the above assumptions. It showed that industrial control systems could now be targeted by malicious code and that *not even hardware-based controllers were safe.* Stuxnet employed a really sophisticated attack mechanism that took control of the industrial automation system executing on a PLC. It took control of the system and operated it according to the attacker's design. It was also able to *hide these changes from the designers/engineers who operate the system.* To achieve these results, Stuxnet utilized a large number of complex methods the most notable of which was the *first known PLC rootkit*. In fact, Stuxnet was present on the infected systems for a long time before it was detected – perhaps even a few months. In this section we will focus on the real target of Stuxnet – the control code that manages the plants and the implications of such an attack.

Stuxnet had the ability to *(a)* monitor blocks that were exchanged between the PLC and computer, *(b)* infect the PLC by replacing legitimate blocks with infected ones and *(c)* hide the infection from designers. The PLCs are used to communicate with and control 'frequency converter drives' that manage the frequency of a variety of motors. The malicious code in the infected PLC affects the operational frequency of these motors so that they now operate outside their safety ranges. *E.g.*, in one instance, the frequency of a motor was set to 1410 Hz, then 2 Hz and then to 1604 Hz and the sequence is repeated – the normal operating frequency for this motor is between 807 Hz and 1210 Hz. Hence, in this instance, Stuxnet's actions can result in *real physical harm to the system.*

Note: Our focus is not on preventing the original intrusion or providing mechanisms to safeguard the Windows machines that are infected. We intend to detect the infection of the control code (on a PLC in this example, but could be any computer that runs it) and mainly safeguard the physical system from coming to harm.

2.2 Automotive Attack Surfaces and Other Examples

Researchers from the University of Washington demonstrated how a modern automobile's safety can be compromised by ma-

[4]See Section 4 for details.

licious attackers [4, 13]. They show how an attacker is able to circumvent the rudimentary security protections in modern automobiles and infiltrate virtually any electronic control unit (ECU) in the vehicle and compromise safety-critical systems (that have stringent real-time properties) such as disabling the brakes, stopping the engine, selectively braking individual wheels on demand, *etc.* – all of this, while ignoring the driver's inputs/actions. They were able to achieve this due to the vulnerabilities in the CAN bus protocols used in many modern vehicles. The attackers also show how malicious code can be embedded within the car's telematics unit that will completely erase itself after causing the crash.

There have been numerous other attacks that infiltrated critical systems *e.g.* wastewater treatment plants [1], NRG generation plants [17], medical devices [14], *etc.*

2.3 Discussion

As these examples show, safety-critical systems can no longer be considered to be safe from security breaches. While the development of cyber security techniques can help alleviate such problems, the real concern is for the control systems and physical plants that can be seriously damaged – often resulting in the crippling of critical infrastructure. Hence, we propose *non-traditional intrusion detection and recovery mechanisms* to tackle such problems. We use to our advantage the fact that the control codes running in a real-time system tend to be deterministic in behavior, simple to implement and exhibit strict timing properties. In fact, our techniques, if used on the above systems, could have gone a long way in mitigating (or at least quickly detecting) the attacks.

For the rest of this paper, we will show how such intrusions can be detected and the harmful effects mitigated by use of our *Secure System Simplex Architecture* (S3A). Hence, *our aim is to identify, as quickly as possible, that an infection has taken place and then ensure that the system (and its physical components) are always safe*. **Note:** as stated in the introduction, our work does not aim to prevent the original infections since that is a large problem that requires the development and implementation of multiple levels of cyber security techniques/research. We focus on the aftermath of the infection of control codes.

3. THREAT MODEL

We deliberately will not delve too deeply into specific threat models, since we believe that our techniques will work well for a broad class of attacks that modify the execution behavior of embedded code in safety-critical systems. Attacks similar to those in Section 2 can be caught by the mechanisms presented in this paper. Hence, code could be injected by any of the mechanisms described in that section – as long as the malicious entity tries to execute *any new code* on the control side, we will be able to detect it. Hence, our threat model [12] is quite broad and can detect attacks such as: *(a) physical attacks*, i.e. code injected via infected/malicious hardware; *(b) memory attacks* where attackers try to inject malicious code into the system and/or take over existing code; *(c) insider attacks* where the attackers try to gain control of the application/system by altering all or part of the program at runtime.

We will, instead, focus on what happens *after* attackers perform any of the above actions in order to execute their code. Hence, we intend to show how our architecture is able to quickly detect this and keep the system(s) safe particularly the physical systems. Since we don't care much about *what* executes and are more concerned with *how long* something executes, our "malicious entity" is a little more abstract as explained later in Sections 5.4.4 and 6.2.

4. SYSTEM SIMPLEX OVERVIEW

The Simplex Architecture [21] utilizes the idea of *using simplicity to control complexity* in order to safely use an untrusted subsystem in a safety-critical control system. A Simplex system, shown in Figure 1, consists of three main components: *(a)* under normal operating conditions the `Complex Controller` actuates the plant; this controller has high performance characteristics and is typically unverifiable due to its complexity; *(b)* if, during this process, the system state becomes in danger of violating a safety condition, the `Safety Controller` takes over; *(c)* the exact switching behavior is implemented within a `Decision Module`. The Simplex architecture has been used to improve the safety of remote-controlled cars [5], pacemakers [2] and advanced avionics systems [19]. Early Simplex designs had all three subsystems located in software – at the application-level. This was updated in *System-Level* Simplex [2] by performing hardware/software partitioning on the system where the safety controller and the decision module are moved to a dedicated processing unit (an FPGA) that is different from the the microprocessor running the complex controller. We leverage this partitioning technique in S3A.

Untrusted Controllers: It is not that designers wish to use unverified (or untrusted) controllers in such systems. Most controllers that are intended to manage anything but the simplest of systems are typically complex and hard to verify. This is especially true if they must also achieve high levels of performance. Hence, there could be bugs and/or potential vulnerabilities in the system that attackers could exploit. Even if we assume that the controller is completely trusted, it can still be compromised (case in point – Stuxnet reprogrammed the controller in the PLC). Our technique can protect against any such intrusion, be it in trusted or untrusted controllers.

System Upgrades: Another issue is what happens if the system must be updated and that process either *(a)* breaks the safety and timing properties of the system or *(b)* introduces malicious code. This is particularly important if such updates were to happen in a remote fashion. While these would be serious issues in most general-purpose or even mobile embedded systems (*e.g.* cell phones), it is not a problem for safety-critical systems since the Simplex architecture has been shown to support upgrades to the complex controllers [20] in a safe manner. Also such systems are rarely updated, if at all. Any potential updates will have the following properties: *(1)* they are never performed remotely and carried

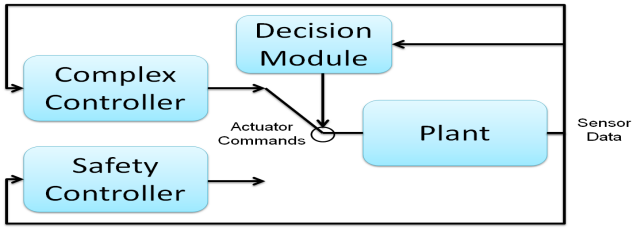

Figure 1: Simplex Architecture

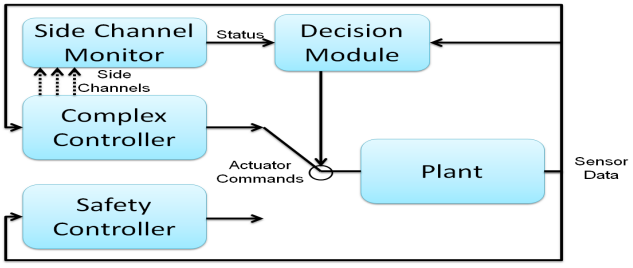

Figure 2: S3A Architecture

out by trusted engineers; *(2)* most updates are minor in that they only tune certain parameters and rarely, if at all, modify the control/timing structure of the code – hence they will not even modify the safety properties of the system and *(3)* major changes, if any, will require extensive redesign, testing, *etc.* – hence the real-time properties of the system must then be re-analyzed anyways.

Discussion: Our application of Simplex in S3A, in this paper, has several significant differences compared with earlier approaches. In the past, the primary motivation to use Simplex was to aid in the verification of complex systems. In this work, we instead apply Simplex to protect against malware that has infected the complex controller. Another key difference is that previously the decision module's behavior was determined completely by the physical state of the plant. In this work, we widen the scope of the "correct state" by using side channels from the computational part of the system, such as the timing properties of executing real-time tasks, in order to determine when to perform the switching. The Simplex decision module is now monitoring *both*, the physical system as well as the cyber state of the computational system.

5. INTEGRATED FRAMEWORK FOR SECURITY: SECURE SYSTEM SIMPLEX ARCHITECTURE (S3A)

We now present the *Secure System Simplex Architecture* (S3A) that prevents damage to safety-critical systems and also aids in rapid detection of malicious intrusions through side-channel monitoring. We first elaborate on the high-level logical framework of the architecture. We then discuss aspects of the execution time-based side channels that we have implemented in our S3A prototype and then follow it up with details on how to implement such a system – from the hardware aspects to the OS modifications; from the timing measurements to the control system that we use to show the effectiveness of our approach.

5.1 High Level Architecture

Figure 2 provides a high level overview of the system architecture. There is a `Complex Controller` that computes the control logic under normal operations. The computed actuation command is sent to the plant and sensor readings are produced and given to the controller to enable feedback control. There is also a `Decision Module` and `Safety Controller` in this architecture that are used not only to prevent damage to the plant in case of controller code bugs (as with the traditional Simplex applications) but also to prevent plant damage in the case of malicious actuation from attackers. We also have a `Side Channel Monitor` that examines the execution of the complex controller for changes in 'expected' behavior (in this paper it monitors the execution time of the complex controller to see if there is any deviation from what is expected). If the information obtained via the side channels differs from the expected model(s) of the system, the decision module is informed and control is switched to the safety controller (and an alarm can be raised). The types of side channels we can consider in a CPU-based embedded system include the execution time profiles of tasks, the number of instructions executed, the memory footprint and usage pattern or even the external communication pattern of the task. We will discuss timing side channels in more detail in the Section 5.2 and elaborate on the viability of the others in Sections 5.3 and 9.

This approach is qualitatively more difficult to attack than a typical control system. An attacker not only has to compromise the main system, but she also has to replicate all side channels that are currently being monitored. If the execution time is being monitored then the attacker must replicate the timing profile of a correctly-functioning system. If the cycle count is being observed, her code must also execute for a believable number of instructions. Even if all the side channels match the expected models, the Decision Module will still monitor the plant state and, when malicious actuation occurs, prevent system damage.

The effectiveness of the side channel early-detection methodology depends on two factors. First, the constructed model of each side channel should restrict valid system behavior (not easily replicable). Second, the side channel itself must be secure (not easily forgeable). These factors are implementation specific and will be discussed later in Section 5.4.

5.2 Timing Side Channels

In this paper, we intend to secure a real-time embedded system. Therefore, we assume that the system has typical real-time characteristics, *i.e.* the system is divided into a set of periodic tasks managed by a real-time scheduler. Each task has a *known execution time* and each task periodically activates a job.

The monitoring module maintains a precise timing model of the system. Violations of this model occur when a job's, *(i)* execution time is too large; *(ii)* execution time is too small; *(iii)* activation period is too large or *(iv)* activation period is too small.

The monitoring module also needs to examine the execution of the *idle task*. This prevents a malicious attacker from allowing the real-time task to execute normally and perform malicious activity during idle time. Finally, the monitoring module should monitor the system activities that may result in timing perturbations.

In our prototype, we *monitor the control task* and the *idle task*. For rapid prototype development, we eliminate system noise (disable interrupts) while our control task is running to obtain a predictable timing environment[5] rather than patching system interrupts in order to receive their timing information. In an actual real-time system interrupts would be predictable and scheduled deterministically – hence we would be able to monitor them as well as the tasks. This addition could be made to our prototype in the future.

Execution times of the various real-time tasks in such systems are anyways obtained as part of system design by a variety of methods [22]. There is no extra effort that we have to perform to obtain this information. The worst-case, best-case and average-case behavior for most real-time systems is calculated ahead of time to ensure that all resource and schedulability requirements will be met during system operation. We use this knowledge of execution profiles to our advantage in S3A.

5.3 Other Potential Time-based Side Channels

In the assumed context of predictable real-time embedded control systems, several other side channels are available as part of the cyber state such as *task activation periodicity, memory footprint, bus access times and durations, scheduler events, etc.*. Each of these is a candidate for *benevolent side-channels* that can be monitored to detect infections and would have to be individually replicated by an attacker to maintain control in an infected system, thus qualitatively increasing the difficulty for such actions.

Additionally, the specific side channels used may vary depending on the type of system. Here, we focus on CPU-based real-time control systems. Other systems, *e.g.* PLC-based ones, would likely need to either monitor the side channels using different methods or utilize a completely different (or additional) sets of side channels.

[5]Details in Section 5.4.5.

5.4 Implementation

We now describe a prototype implementation of S3A that we have created. The technical details of the prototype are listed in Table 1. We elaborate on key aspects of our implementation in detail: first, a hardware component overview is provided in Section 5.4.1. Then, the inverted pendulum hardware (our example 'safety-critical control system') setup is described in Section 5.4.2. The methodology for timing measurements of the control code is described in Section 5.4.3 and the methodology for timing-variability ('malicious code') tests is presented in Section 5.4.4. Section 5.4.5 gives essential details about the operating system setup during the measurements. Finally, Section 5.4.6 describes the specific design of the Decision Module and the timing Side Channel Monitor.

Component	Details
Inverted Pendulum	Quanser IP01
FPGA	Xilinx ML505
Computer with Controller	Intel Quad core 2.6 GHz
Operating System	Linux kernel ver. 2.6.36
Timing Profile	Intel Timestamp Counter (rdtsc)

Table 1: S3A Prototype Implementation Details

5.4.1 *Hardware Components*

A high-level hardware design of our prototype is shown in Figure 3. The prototype hardware instantiates the logical Secure System Simplex architecture previously described in Section 5 and shown in Figure 2. In our implementation, we run the complex controller on the main CPU. The Complex Controller communicates with a trusted hardware component, an FPGA in this case, to perform control of an inverted pendulum. Sensor readings are obtained by the FPGA over the PCIe bus using memory mapped I/O. The actuation command, in turn, is written to the memory-mapped region on the FPGA. Additionally, timing messages in the form of memory-mapped writes are periodically sent to the FPGA based on the state of execution (at the start/end of the control task and periodically during the Idle Task). This creates a timing side channel that can be observed by a timing channel monitor running on the FPGA. On the FPGA side, the timing channel monitor will measure the time elapsed between timing messages from the complex controller to ensure that the execution conforms to an expected timing model. The decision module will periodically examine the output of the Timing Channel Monitor, the actuation command from sent by the Complex Controller from shared memory on the FPGA, the actuation command from the locally-running safety controller and the state of the plant from a 'sensor and actuator interface' and decide which controller's actuation command should be used – the complex one on the CPU or the safe one on the FPGA. The actuation command is then output back to the Sensor and Actuator Interface.

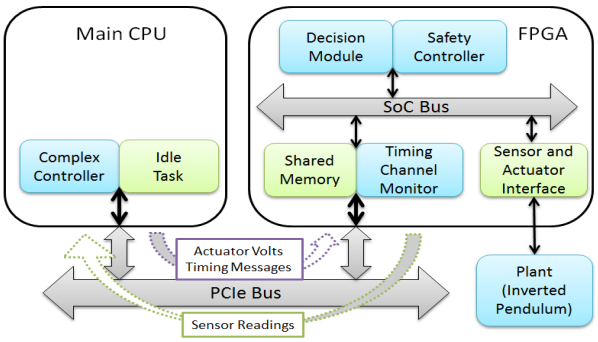

Figure 3: S3A Implementation Overview

The interface then, through a digital-to-analog converter, actuates the plant – in our case, an inverted pendulum. The Sensor and Actuator Interface also periodically acquires sensor readings through analog-to-digital converters and write their values to both shared memory accessible by the Complex Controller and to memory accessible by the trusted Decision Module and Safety Controller.

5.4.2 *Inverted Pendulum*

We used an inverted pendulum (IP) as the plant that was being controlled. An IP (*e.g.* Figure 4) is a classic real-time control challenge where a rod must be maintained in an upright position by moving a cart attached to the bottom of the IP along a one-dimensional track. It has two sensors (to measure the current pendulum angle and the cart position on the track) and one actuator (the motor near the base of the pendulum) used to move the cart. Two safety invariants must be met: *(1)* the pendulum must remain upright (can not fall over) and *(2)* the cart must remain near the center of the track. The specific inverted pendulum we used in our testbed was based on the Quanser IP01 linear control challenge [9].

Our setup varies slightly from an off-the-shelf Quanser IP01 as follows: we need to directly connect the sensors and actuators to the FPGA; the prebuilt setup requires a computer to do the data acquisition. We modified the system to redirect the sensor values and motor commands through an Arduino Uno microcontroller that communicates directly with the S3A FPGA through a serial cable. Although this may introduce latency into the system, we did not observe any issues with safely actuating the pendulum due to this small delay. The control code that manages the IP executes on a computer (Section 5.4.5 and Table 1) at a frequency of 50 Hz.

Note: The IP has been used quite extensively in literature as an appropriate example of a real-time control system [2,21]. Hence we believe it demonstrates an early prototype (and proof-of-concept) of our solutions. We are currently working on applying these techniques to other real control systems in conjunction with industry.

5.4.3 *Timing*

The implementation of the complex controller for the inverted pendulum is fairly simple with very few branches and most loops being statically decidable[6]. Hence it is fairly easy to calculate the execution time and number of instructions taken for such code. In our framework, we utilized simple dynamic timing analysis [22] methods to obtain an *execution profile* of the code. We used the Intel *time stamp counter* (`rdtsc`) [10] to obtain high resolution execution time measurements for the control code.

The control code was placed in a separate function and called in a loop. As part of our experiments, the loop was executed $1, 10, 100, 1,000, 10,000, 100,000$ and $1,000,000$ times on the actual computer where it would execute and measuring each set of executions. During each of these scenarios, the total time of the loop as well as the times taken up during each individual iteration

[6]This is typical of most control code in safety-critical and real-time control systems – hence our implementation of the controller for the inverted pendulum is also similar.

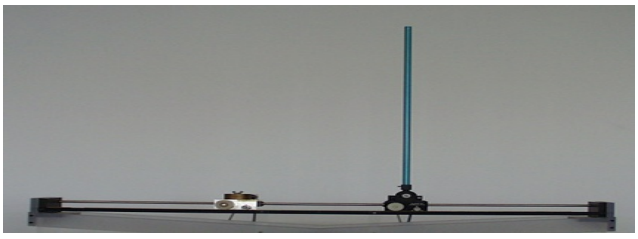

Figure 4: An inverted pendulum control system

was measured. From these traces we were able to determine the maximum (worst-case), minimum (best-case) and steady-case values for the execution time of the controller code. 'Steady-case' refers to the values obtained when the execution time has stabilized over multiple, repeated executions – *i.e.* when the initial cold cache related timing dilation at the start of the experiments no longer occur. To reduce the noise from instrumentation and overheads of the loops, function calls, *etc.* we used the 'dual-loop timing' method: *i.e.*, empty loops with only the measurement instrumentation were timed as a 'control' experiment. The execution times obtained for these instrumentation-only loops were subtracted from the execution times for the loops with the control code. While we used simple measurement-based schemes for obtaining the execution profile in this paper, it does not preclude the use of more sophisticated techniques [16, 22] to obtain better (and safer) timing estimates. This is especially true if the code is more complex than the one for the inverted pendulum. In fact, the better the estimation methods, the better S3A will be able to detect anomalies and intrusions.

Interrupts (all interrupts including inter-processor ones) were disabled during timing measurements. To reduce the effects of the operating system and other system issues we isolated our controller as best as we could as we will describe in Section 5.4.5.

5.4.4 Execution Time Variation

To mimic the effect of code modification on timing, we insert extra code into the execution of the control loop function described above. Specifically, the extra code is a loop with a varying upper bound (*i.e.* $1, 10, 100$) that performs multiple arithmetic operations (floating point and integer). The idea is that the extra instructions that execute will make it look like an intrusion has taken place. Our S3A system will then detect the additional execution, raise an alarm and transfer control to the simple controller on the FPGA.

Note: As mentioned before, we are less interested in what kind of code executes "maliciously" because our detection does not depend on this detail. We only need to check whether whatever is executing has modified the timing profile of the system.

5.4.5 System and OS Setup

We used an off-the-shelf multi-core platform running Linux kernel 2.6.36 for our experiments (Table 1). Since we use a COTS system, there are many potential sources of timing noise such as cache interference, interrupts, kernel threads and other processes that must be removed for our measurements to be meaningful. In this section we describe the configuration we used to best emulate a typical uni-processor embedded real-time platform.

The CPU we used is an Intel Q6700 chip that has four cores and each pair of cores shares a common level two (last level) cache. We divided the four cores into two partitions: **1.** the *system partition* running on the first pair of cores (sharing one of the two L2 caches)

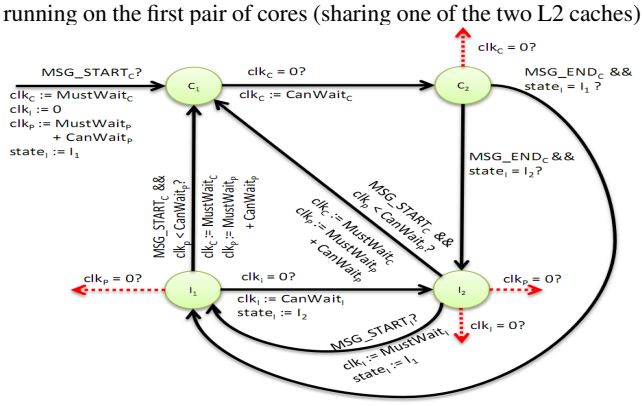

Figure 5: FSM for Detecting Timing Model Violations

handles all interrupts for non-critical devices (e.g., the keyboard) and runs all the operating system activities and non real-time processes (*e.g.*, the shell we use to run the experiments); **2.** the *real-time partition* runs on the second pair of cores (sharing the second L2 cache). One core in the real-time partition runs our real-time tasks together with the driver for the trusted FPGA component; the other core is turned off so that we avoid L2 cache interference among these two cores.

5.4.6 Detection

In our system, detection of malicious code can occur in one of two ways. The decision module observes both *(i)* the physical state of the plant (by traditional Simplex) as well as *(ii)* the computation state of the system (based on timing messages; S3A). A violation of the physical model or the computational model can trigger the decision module to transfer control to the safety controller on the FPGA. Based on a function of the track position and pendulum angle (the physical model), the decision module may choose to switch over to the safety controller [2, 21].

The computational system is monitored for violations of expected timing model of the system. Both, the control task as well as the idle task, are monitored in order to periodically send timing messages to the FPGA that contains an *expected timing model of the system as a finite state machine* (FSM) *running in hardware*. When timing messages arrive (or timers expire) the FSM advances. If malicious code were to execute, it would have a limited window of time to replicate the timing side channel before it was detected by the decision module.

Generally speaking, monitoring the timing progress of a real-time system can be performed by maintaining state about each task in the system. A task has two timers associated with it: **(I)** the first would enforce the execution time of the task and **(II)** the second will monitor periodic activation of the task. A stack is used to track task preemptions. Since typical real-time systems use priority-based execution, all task switches are directly observable by the FPGA through task start/task end messages.

For our specific prototype, we implemented the finite state machine (Figure 5), in hardware, on an FPGA. Our system contains two tasks: *(i)* the idle task and *(ii)* the controller task. Since only one task may be preempted (the idle task), we maintain a single variable as the call stack, state_I. Three timers are used: clk_C and clk_P maintain the execution time and period of the control task while clk_I maintains the execution of the idle task. In Figure 5, clk_C ticks while the control task is running (states C_1 and C_2) and clk_I ticks while the idle task is executing (states I_1 and I_2). Clk_P always ticks. The FSM is parameterized with six values: MustWait_C, CanWait_C, MustWait_I, CanWait_I, MustWait_P, and CanWait_P. These values are determined by the minimum and maximum time permitted between timing messages. The MustWait time indicates the minimum time that must elapse, whereas the CanTime indicates the jitter permitted between different iterations of the loop. Hence, MustWait is the minimum execution time of the task/idle loop/period whereas (MustWait + CanTime) is the maximum execution time.

In the FSM, initially the control task is running. State C_1 is entered and continued in until clk_C ticks from MustWait_C to 0. Then state C_2 is entered. If clk_C ticks from CanWait_C to 0 without the end task message then the control task has executed for too long and a timing violation occurs (indicated by dotted arrow in state C_2). Once the end control task message is received, the idle task begins to execute. Under normal operation, the state changes between I_1 and I_2 several times, until the control task is reactivated and state C_1 is again entered. Any messages that arrive without explicit tran-

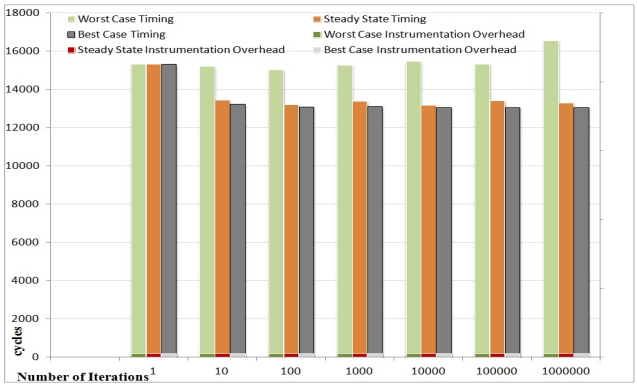

Figure 6: Summary of the Timing Results

sitions in the FSM are interpreted as errors in the prototype and trigger the decision module to switch to the safety controller. Additionally, dotted transitions in the FSM are timing violations that also trigger the decision module to take corrective action.

The FSM can also be used to tightly track the execution behavior of the code for more sophisticated controllers, *e.g.* if the code has many branches, function calls, *etc.* For instance, when the control code reaches a branch that affects the overall execution time, a message can be sent to the FSM about which side of the branch was taken. The FSM can now use this information to accurately track the execution of the program for all control constructs in the code.

6. EVALUATION

We now present an evaluation of S3A – Sections 6.1 and 6.2 present timing results obtained by analysis of the controller code – these values form the profile of the execution behavior used in the intrusion detection mechanism on the FPGA. Section 6.3 presents the details of the intrusion detection.

6.1 Timing Results and Execution Profile

Figure 6 presents a high level summary of the timing results used to obtain the execution profile of the complex controller code (Figure 2). We used dynamic/run-time timing analysis techniques to obtain the worst, best and steady state execution times for this code. The x-axis represents the number of times the controller code was repeatedly executed: from 1 to $1,000,000$ in steps of 10. The y-axis represents the execution time in *cycles*. Each grouping of vertical lines represents the 'worst-case', 'steady-state' and 'best-case' execution times for that experiment. 'Steady-state' refers to the execution time when successive executions of the controller code resulted in the same execution time – *i.e.* the situation when the execution reached a steady state. The 'worst-case' numbers in the graph are usually different from the first few iterations before the system effects (in particular the cache) have settled down. This is the reason why there exists a slightly larger difference between the worst-case and best-case numbers.

Each vertical bar is split into two parts – the lower part shows the instrumentation overhead for that experiment[7], while the top part represents the timing for the control code only. We also see that the instrumentation overhead is almost the same across all experiments – oscillating between 260 and 270 cycles for all experiments.

As seen in the graph, the steady state and best-case values are very close, not just within the same experiment, but across experiments. The largest difference between the two is 360 cycles for the $n = 100,000$ experiment. This just shows that our assumption that controller codes in safety-critical systems are simple and have little variability is valid. This lack of variability is also evident from the fact that the worst-case execution cycles, across experiments, do not show much variance. The worst-case values for the last experiment $(1,000,000)$ has a slightly higher value of $16,560$ and this could be due to the initial cold cache and other system effects.

Figure 7 shows the execution profile for one timing experiment in particular – that of $100,000$ iterations. The x-axis is the iteration number while the y-axis is the number of cycles for each iteration. As this figure shows, the first few iterations take a little longer (around $17K$ cycles) and then most of the execution stabilizes to within a narrow band of:

$$1,590\ cycles = 14,660 - 13,070$$
$$i.e.\ \sim 0.6\ \mu s\ \ at\ \ 2.67\ GHz$$

This band defines the '*accepted range*' of values that the FPGA uses to check for intrusions. Any execution that changes the steady state execution time by more than this narrow range will be caught by the FPGA. In fact, the FPGA will catch variance in either direction – *i.e.* increase/decrease in execution time.

The graph also shows that while the majority of execution times fall within a small band at the lower end of the above mentioned range, some values also fall into a narrow band at the top of the range (*i.e.* around the $14K$ value). This narrow band of increased execution times is due to latent system effects that we were not able to remove. The main culprit is the last level cache that, in this architecture, uses a random replacement policy. Hence, every once in a while a few of our controller's cache lines are evicted by periodic kernel threads that we could not disable (since we are running a COTS operating system) and these iterations take a few hundred cycles extra (anywhere from $500 - 900$) to execute. With a more predictable cache replacement policy, like the ones used in hard real-time systems, we would not see this behavior. To prove this theory we ran the same experiments on a PowerPC with pseudo-LRU (Last Recently Used) cache replacement policy in its last level and all the points are clustered into a single band. In fact,

[7]As explained in Section 5.4.3, we used dual-loop timing techniques to obtain the overheads due to the instrumentation.

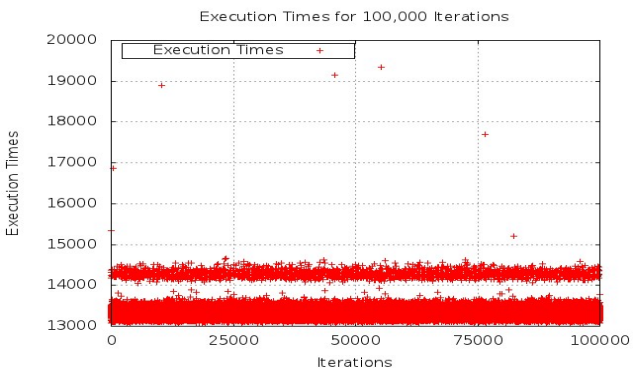

Figure 7: Execution Profile (100,000) with FPGA

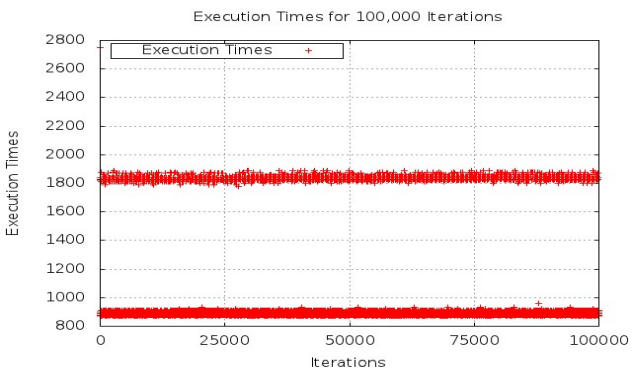

Figure 8: Execution Profile (100,0000) without FPGA

with LRU, tasks would not evict each other cache's lines unless the cache is not big enough to fit them at the same time[8]

Figure 7 also shows a few sporadic experiments exhibiting much higher execution times. Again, this is due to system effects and in particular, contention on the bus when communicating with the FPGA. The complex controller reads and writes messages to and from the FPGA to control the pendulum and to send the timing messages (Section 5.4.1). Many times, when the complex controller is waiting for data from the inverted pendulum that arrives on the common bus, the incoming messages experience unpredictable delays. These delays are due to bus contention among the FPGA and other peripherals sharing the same bus. To prove that the communication with the FPGA was the cause of these effects, we conducted timing experiments where the FPGA was switched off and all calls to communicate with it (read/write) resulted in null function calls. Figure 8 shows the results of these experiments for the $100,000$ iterations point. This experiment highlights two important points: *(1)* the random spikes at higher values no longer exist, thus showing that the bus contention due to communication with the FPGA was the main cause of the spikes; and *(2)* the same 'double-band' of execution results appears here; interestingly, the gap between the bands is almost identical to that of Figure 7, thus providing more evidence to the fact that the cache (and its replacement policy) is the culprit.

Such issues could be avoided when using actual hard real-time systems instead of the COTS-based experimental setup that we use here. In fact, a hard real-time system would use a more predictable bus, or other techniques [3], that allows designers to bound I/O contention and avoid random spikes.

6.2 Malicious code Execution Results

We introduce "malicious code" by inserting extra instructions (Section 5.4.4) – *i.e.* a loop of variable size within the complex controller code. The upper bounds for the malicious loop are one of $1, 10, 100$ – we stopped at the upper limit of 100 since anything over this value would put the execution of the "infected" control code over the real-time period of the task. Also, as we will see soon, even these small additional increases in execution times are caught by S3A.

Figure 9 shows the execution time (in cycles, on the y-axis) taken by the code for value of the malicious loop values (x-axis). The final bar in the graph represents the "base," *i.e.* the number of execution cycles taken up by the controller code without any malicious loop. As expected, the values for the malicious code increases significantly with each increase in the loop bound. Even the smallest sign of the presence of the malicious loop puts it outside of the narrow range ($0.6\mu s$) explained in Section 6.1. Hence, even this will be caught by S3A and control will be transferred over to the simple controller executing on the FPGA. **Note:** Since we don't really care *what* executes as part of malicious code and *intend to only catch variations in execution time*, we only mimic the increased execution time effects by the methods discussed in this section.

[8]If this is the case, we just have to account for it, when we compute the execution time for each task.

Figure 9: Execution Cycles for Malicious Code execution

6.3 Intrusion Detection

We now describe the evaluation of our timing side channel intrusion detection technique. First we describe timing measurements for obtaining key aspects of our architecture. Then, we demonstrate the *early detection of malicious code execution* using the timing side channel approach (S3A) and compare it with monitoring the plant state only (vanilla Simplex). The results for the inverted pendulum are summarized in Table 2.

Our first timing measurement was to obtain the *overheads for sending timing messages* to the FPGA. Although the message itself takes time to propagate through the PCIe bus to the FPGA, the CPU is not stalled during this time. By using the time stamp counter we measured the overhead on the CPU for sending a single timing message to be 130 cycles (50 nanoseconds). This time is extremely small and therefore each process could realistically send multiple messages during a single iteration of each control loop to reduce the time an attacker has to replicate the timing side channel. Another advantage of having multiple timing messages per iteration is that if the program contains branches, we could communicate to the FPGA timing monitor (at run-time) information about which branch was taken thus allowing for tighter monitoring of the timing requirements in the timing model FSM.

The second timing measurement was to quantify the *jitter of the timing messages* going to the FPGA(through the interconnect). We recorded the difference between the arrival of the start and end control iteration timing messages, in the FPGA, over several thousand iterations of the control loop. The reason for this jitter is twofold: *(a)* jitter of the execution time itself (the difference between the minimum and maximum execution time (Figure 5) and *(b)* varying time of message propagation through the PCIe bus.

Since our testbed is an off-the-shelf multicore system (with Linux), processes running concurrently on other cores as well as other independent bus masters (*e.g.* peripherals) may cause interference on the shared interconnect. In a deployed real-time control system, such noise would not be present or would at least be bounded. Nonetheless, we measured the typical timing variation caused by the interconnect to be about 0.6 microseconds, or less than $\frac{1}{8}^{th}$ *of the iteration time of a single control iteration*. The FPGA timing can now detect an intrusion using the timing-based side channel within $5.7\mu s$ and anything that changes the timing by $0.6\mu s$ would be caught (Table 2). We could add multiple timing messages in each control iteration (asthe CPU message overhead is so low) to further reduce the maximum intrusion detection delay.

The control task execution time (Table 2) was obtained from the execution time measurements in Section 6.1. The values are in absolute times that were converted from our cycle count measurements. Hence, the $4.6--5.7\mu s$ value for the 'Control Task Execution Time' is obtained from the (approx.) $13,000 - 14,000$ cycles that we discussed in Section 6.1 and Figure 7.

Due to the extra jitter caused by the interconnect, the enforced iteration time is expectedly larger than the measured control task ex-

Measured Quantity	Time (μs)
Control Task Exec. Time (single iter)	4.8 - 5.4
Interconnect Extra Jitter	$\sim$ 0.6
Enforced Iteration Time	4.6 - 5.7
Timing Anomaly Detection Time (for IP)	5.7
Vanilla Simplex Anomaly Detection Time	10,000
Timing Message CPU Overhead	0.05

Table 2: Measured Timings during Intrusion Detection

ecution time. The maximum enforced iteration time, 5.7μs, is the maximum time the experimental framework can proceed without receiving a timing message before the safety controller takes over. Hence, in the FSM (Figure 5), the runtime value of $MustWait_C$ is 4.6μs, and the runtime value of $CanWait_C$ is about 1.1μs ($mustWait_I$ and $canWait_I$ are much lower). Given these numbers, the side-channel monitor FSM will detect a missed timing message within 5.7μs, *i.e.* the detection time reported.

We now compare the early detection of malicious code through timing side channels with the situation when only the plant state is being monitored (vanilla Simplex). In the timing side channel version, as discussed above, the maximum time that can proceed before without valid timing messages is 5.7μs. For vanilla Simplex, we experimentally measured the amount of time needed to detect an intrusion. After taking control of the system, we tried to destabilize the pendulum by sending a maximum voltage value in the direction that would most quickly collapse the pendulum (in order to obtain a lower bound on the detection time when plant state is monitored alone). For vanilla Simplex, we were able to detect an intrusion after 5 control iterations, or 100 milliseconds. Hence, the use of timing side channels enables significantly faster detection of security vulnerabilities in real-time control systems: over four orders of magnitude faster than with traditional Simplex.

This test shows that if a smart attacker is able to override all of our checking mechanisms (say, by gaining root access) then *the physical system will still remain safe if she tries to destabilize it, since the base Simplex mechanism will kick in and take control*.

7. LIMITATIONS

S3A is not meant to be a silver bullet for intrusion detection in embedded control systems and does have some practical restrictions that may limit its applicability. First, to use S3A in a real system, the latter needs to be designed with the architecture in mind. While this is a limitation for some existing systems, we think that future architectures could provision for such techniques since it is never a bad idea to consider security while designing a new system.

One concern is making sure that an attacker cannot easily replicate our side channels. This could be overcome with minor modifications to the processor architecture or, in our prototype, allowing the FPGA to directly access the instruction count without explicit communication from the CPU.

Additionally, for each side channel, a model of the correct behavior must be created that restricts a malicious program. For the timing side channel, one problem could be that the execution times may have too much variability. While this is possible in general purpose systems, it is not very likely in real-time systems. Even so, this could be overcome at runtime by having each timing-behavior-modifying branch point send information to the FPGA indicating what path was taken, resulting in tight bounds on execution time. The construction and tuning of the timing parameters of the state machine is currently a manual process. We believe this could eventually become an automated step by performing a compile-time analysis of the control flow graph of the code combined with runtime analysis to perform precise timing measurements.

The implementation of the FPGA hardware in our framework must be correct for the system to be secure. This may seem like we have just moved the problem over to securing the FPGA system but this is not the case since: the FPGA and Safety Controller only need to maintain the safety of the plant. The Complex Controller, on the other hand, can perform useful work with the plant so any upgrades will be made to the Complex Controller and not to the FPGA's safety logic. Of course, we should not permit FPGA reconfiguration at runtime. One other issue related to the use of FPGAs floating-point computation units are typically not present since they use up significant area. The FPGA in our architecture is used as a rapid prototype of the trusted simplex component. A deployment implementation could use a trusted microcontroller along with any capabilities (*e.g.* floating point units) that are needed for the various components. Also, the FPGA will only host the safety controller that maintains bare functionality. Hence, it is unlikely that it will need to perform fancy floating point calculations.

The original Simplex only protects systems from properties that are known to result in unsafe states. *E.g.* in Stuxnet, the malicious controller would actuate the plant motor for periods at very high frequencies and then for periods at very low frequencies in order to damage the motors. If the Decision Module was not monitoring this property, such unsafe actuation would still proceed to the plant.

8. RELATED WORK

Zimmer *et. al.* [24] use worst-case execution time (WCET) information to detect intrusions by instrumenting the tasks and schedulers with periodic checks on whether the execution has gone past expected WCET values. We focus on detecting intrusions in real-time control systems and ensuring that the plant remains safe even if the intruder is able to bypass all detection/security mechanisms. As compared to them, we ensure that the system remains safe even if an intruder gains root privileges to the system. Our monitoring is performed by a trusted hardware component, separate from the main system, thus increasing the robustness of the architecture.

The trusted computing engine (TCE) [11] and the reliability and security engine (RSE) [12] also use secure co-processors to execute security-critical code/monitor the access of critical data. We don't require the information about what data is critical or even touch the source code. We detect intrusions by observing the innate characteristics of the program at runtime.

The IBM 4758 secure co-processor could be used to perform intrusion detection [23]. This work contains a CPU, separate memory (volatile and non-volatile) along with cryptographic accelerators and comes wrapped in a tamper-responding secure boundary. While we could adapt this processor for use with our architecture, the main difference from S3A lies in the fact that we employ the inherent characteristics of the program to detect intrusions, especially in the CPS domain; also coupling with the System Simplex mechanism increases the robustness of the overall system. FlexCore [6] uses a reconfigurable fabric to implement monitoring and book-keeping functions. Compared to FlexCore, we *(a)* don't need to know what types of attacks are taking place (as long as it modified the execution time behavior of our code) and *(b)* don't need to analyze the program structure/data.

Other related work includes Pioneer [18] (sophisticated checksum code and execution time information to establish safe remote execution on an untrusted computer), TVA [8] (provides guarantees that the software running on a general purpose computer is intrusion-free in conjunction with a hardware trusted component, TPM), and PRET [15] ('precision timed machines' to detect and protect against side-channel attacks). Our work is different from all of these in significant ways, since we don't touch actual code, try to protect local control systems with real-time properties and use side-channels to our benefit.

9. CONCLUSIONS

We presented a new framework, Secure System Simplex Architecture (S3A), that enhances the security and safety of a real-time control system. We use a combination of trusted hardware, benevolent side-channels, OS techniques and the intrinsic real-time nature

(and domain-specific characteristics) of such systems to detect intrusions and prevent the physical plant from being damaged. We were able to detect intrusions in the system in less than 6 μs and changes of less than 0.6 μs – time scales that are extremely hard for intruders to defeat. We show that even if an attacker is able to bypass all security/intrusion detection techniques, the actual plant will remain safe. Another important characteristic of these techniques is that there are *no modifications required in the source code*. We believe that the novel techniques and architecture presented here will significantly increase the difficulty faced by would-be attackers thus improving the security and overall safety of such systems.

10. REFERENCES

[1] ABRAMS, M., AND WEISS, J. Malicious control system cyber security attack case study – maroochy water services. http://crc.nist.gov/groups/SMA/fisma/ics/documents/Maroochy-Water-Services-Case-Study_report.pdf, 2008.

[2] BAK, S., CHIVUKULA, D. K., ADEKUNLE, O., SUN, M., CACCAMO, M., AND SHA, L. The system-level simplex architecture for improved real-time embedded system safety. In *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 99–107.

[3] BETTI, E., BAK, S., PELLIZZONI, R., CACCAMO, M., AND SHA, L. Real-time i/o management system with cots peripherals. *Computers, IEEE Transactions on PP*, 99 (2011), 1.

[4] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security* (Aug 2011).

[5] CRENSHAW, T. L., GUNTER, E., ROBINSON, C. L., SHA, L., AND KUMAR, P. R. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 400–412.

[6] DENG, D. Y., LO, D., MALYSA, G., SCHNEIDER, S., AND SUH, G. E. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 137–148.

[7] FALLIERE, N., MURCHU, L., AND (SYMANTEC), E. C. W32.stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2011.

[8] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 193–206.

[9] INDUSTRIAL, Q. Inverted pendulum [ip] linear. Quanser IP01, 2011.

[10] INTEL. Using the RDTSC instruction for performance modeling. www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[11] IYER, R. K., DABROWSKI, P., NAKKA, N., AND KALBARCZYK, Z. Reconfigurable tamper-resistant hardware support against insider threats: The trusted illiac approach. 133–152. 10.1007/978-0-387-77322-3_8.

[12] IYER, R. K., KALBARCZYK, Z., PATTABIRAMAN, K., HEALEY, W., HWU, W.-M. W., KLEMPERER, P., AND FARIVAR, R. Toward application-aware security and reliability. *IEEE Security and Privacy 5* (2007), 57–62.

[13] KOSCHER, K., CZESKIS, A., ROESNER, F., PATEL, S., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., AND SAVAGE, S. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on* (may 2010), pp. 447 –462.

[14] LI, C., RAGHUNATHAN, A., AND JHA, N. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *e-Health Networking Applications and Services (Healthcom), 2011 13th IEEE International Conference on* (june 2011), pp. 150 –156.

[15] LIU, I., AND MCGROGAN, D. Elimination of side channel attacks on a precision timed architecture. Tech. Rep. UCB/EECS-2009-15, EECS Department, University of California, Berkeley, Jan 2009. This a class project report describing early work on eliminating side channel attacks using PRET.

[16] MOHAN, S., AND MUELLER, F. Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *IEEE Real-Time Embedded Technology and Applications Symposium* (2008), pp. 285–294.

[17] (NERC), N. A. E. R. C. Jan-june 2009 disturbance index. http://www.nerc.com/files/disturb09-January-June.pdf, 2009.

[18] SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review 39*, 5 (2005), 1–16.

[19] SETO, D., FERREIRA, E., AND MARZ, T. F. Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis). Technical Report Cmu/ sei-99-Tr-020.

[20] SETO, D., KROGH, B., SHA, L., AND CHUTINAN, A. Dynamic control system upgrade using the simplex architecture. *IEEE Control Systems 18*, 4 (Aug. 1998), 72–80.

[21] SHA, L. Using simplicity to control complexity. *IEEE Softw. 18*, 4 (2001), 20–28.

[22] WILHELM, R., AND ET AL. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems 7*, 3 (Apr. 2008), 1–53.

[23] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure coprocessor-based intrusion detection. *Proceedings of the 10th workshop on ACM SIGOPS European workshop beyond the PC EW10* (2002), 239.

[24] ZIMMER, C., BHATT, B., MUELLER, F., AND MOHAN, S. Time-based intrusion detection in cyber-physical systems. In *International Conference on Cyber-Physical Systems* (2010).