COMPACT INTEGRITY-AWARE ARCHITECTURES

BY

MICHAEL D. LEMAY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Abstract

Malware often injects and executes new code to infect hypervisors, OSs and applications on a wide range of systems, from embedded systems to servers in data centers. In this dissertation, we design and evaluate approaches for remotely attesting software integrity and blocking infections on a variety of systems using *integrity kernels*. Existing hardware architectures provide inadequate support for integrity kernels. Despite this, we equip commodity embedded systems with compact integrity kernels. We also describe the limitations of existing non-embedded processors. Then, we develop an extended processor architecture that provides superior isolation, visibility, performance, and compatibility for integrity kernels.

We were the first to demonstrate practical remote attestation for Advanced Metering Infrastructure (AMI), a core technology in emerging smart power grid systems that requires integrity guarantees for each meter over an interval of time rather than just at a given instant. Our prototype *Cumulative Attestation Kernel (CAK)* uses less than one quarter of the memory available on 32-bit Atmel AVR32 flash MCUs similar to those used in AMI deployments. We analyze one of the specialized features of such applications by constructing the first formal proof that security requirements are met by a system even when it experiences unexpected, repeated halt conditions, specifically concerning our prototype. We also developed the only remote attestation mechanism for 8-bit Atmel AVR microcontrollers that communicate over networks like those in AMI and that run untrusted application firmware that can be remotely upgraded.

We created the *Integrity-Aware Processor (IAP)*, which is the only processor architecture with direct support for detecting attempts to execute unverified code. Using the IAP as a base, we developed the smallest integrity kernel that checks all code that ever executes in a target Linux system. It uses a network-hosted whitelist.

Soli Deo Gloria

Acknowledgments

This work was supported in part by DOE DE-OE0000097, NSF CNS 09-64392, NSF CNS 09-17218, NSF CNS 07-16626, NSF CNS 07-16421, NSF CNS 05-24695, NSF CNS 05-24516, NSF CNS 05-09268, NSF CNS 05-5170, ONR N00014-08-1-0248, ONR N00014-04-1-0562, ONR N00014-02-1-0715, DHS 2006-CS-001-000001, HHS 90TR0003-01, and grants from the MacArthur Foundation, Boeing Corporation, and Lockheed Martin Corporation. Michael LeMay was supported on a National Defense Science and Engineering Graduate fellowship from the Air Force Office of Scientific Research for part of this work. The views expressed are those of the authors only.

My advisor, Professor Carl A. Gunter, gave me the freedom to discover and explore my true research passions, even when they fell outside of his core emphases. His intellectual agility enabled him to generate pivotal ideas in our projects regardless of their technical focus. He has actively encouraged and helped me to become a member of the research community and has supported me in my efforts to take our research beyond academia and make it relevant in broader contexts. He encouraged me to take leadership roles soon after I arrived and later challenged me to be a more effective teacher. Finally, he works hard to foster community among the members of his laboratory, both inside and outside of the workplace.

Professor Samuel T. King has been a great source of help and advice throughout my time as a student. I also want to thank the other members of my committee, Professor Sarita V. Adve and Doctor Peter G. Neumann, for their valuable feedback on this work during my preliminary exam, final defense, and on other occasions.

I have greatly enjoyed my collaborations with my labmates, Doctor Jianqing Zhang and Doctor Omid Fatemieh, as well as our discussions and social gatherings. It has been an honor and a privilege to have all of the members of the Illinois Security Lab as associates and friends.

The Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) project as well as its predecessor, the TCIP center, were both critical resources that enabled me to perform inter-disciplinary research with a group of diverse researchers and that provided me with invaluable connections to industry and government.

My wife, Elizabeth, tirelessly and patiently supported me as I completed my dissertation research. Her love and encouragement helped me to overcome the challenges that the project entailed. I also want to thank my parents, David and Susan LeMay, for their support, encouragement, and advice throughout my education. A special thanks is due to my father for bringing me to his workplace early in my life and demonstrating the joys of engineering to me.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AMI	Advanced Metering Infrastructure
CAC	Cumulative Attestation Coprocessor
CAK	Cumulative Attestation Kernel
CCM	Counter-CBC MAC (cryptographic mode)
CESIum	Cumulative Embedded System Integrity
CFI	Control-Flow Integrity
CRAESI	Cumulative Remote Attestation for Embedded System Integrity
CTR	Counter (cryptographic mode)
DMA	Direct Memory Access
DMM	Digital Multi-Meter
ECB	Electronic Codebook
ECC	Elliptic-Curve Cryptography
ECDH	Elliptic-Curve Diffie Hellman
FAN	Field-Area Network
HAN	Home-Area Network
IAP	Integrity-Aware Processor
IED	Intelligent Electronic Device
IO	Input/Output
JIT	Just-In-Time
LRU	Least-Recently Used

LTL	Linear Temporal Logic
MAC	Message Authentication Code
MCU	Microcontroller Unit
MMU	Memory Management Unit
MPU	Memory Protection Unit
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
OS	Operating System
PCR	Platform Configuration Register
PCS	Process Control System
RAM	Random Access Memory
ROM	Read-Only Memory
SMM	(Intel) System Management Mode
SMRAM	System Management RAM
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
TOCTTOU	Time Of Check To Time Of Use
TOUTTOC	Time Of Use To Time Of Check
TPM	Trusted Platform Module
VM	Virtual Machine
WDT	Watchdog Timer
XIVE	eXecuting $\rightarrow$ Verified Enforcer
XNP	XIVE Network Protocol
XST	Xilinx Synthesis Tool

Chapter 1

Introduction

1.1 Motivation

Hypervisors, OSs, and applications continue to be infected by malware [37]. One common result of compromise is the execution of foreign code on the target system. Foreign code can be injected directly into a process as a result of a memory corruption bug, or it can be a separate program that is downloaded and installed on the machine as a part of the attack. Detecting this foreign code is tantamount to detecting the attack. Eliminating this foreign code would severely limit a successful attack.[1]

Remote sensor networks are becoming widespread in critical infrastructure. The networking of these systems often enables firmware to be updated in the field to correct flaws or to add functionality. This capability could potentially be exploited to install malware. A good example of this trend is in the deployment of Advanced Metering Infrastructure (AMI), a centerpiece of "smart grid" technology in which networked power meters are used to collect, process, and transmit electrical usage data, and relay commands from utilities to intelligent appliances. Meters are required to support remote upgrades [1]. Attackers are likely to attempt to compromise the upgrade functionality on AMI devices, since meters have historically been common targets of adversaries seeking to steal electricity [5]. AMI devices may also attract new classes of attackers, given the potential for attacks on AMI to induce large-scale effects. For example, coordinated attacks on demand response systems that use signals from meters to control appliances such as air conditioners may result in blackouts, since the US power grid may operate with a delicate balance between generation and load [17].

It is a best practice to prevent unauthorized firmware (including malware) from being installed on such systems by requiring firmware updates to be digitally signed

[1]This chapter includes material from previous publications by LeMay and Gunter [43, 44].

by a trusted authority. However, the principle of defense-in-depth instructs us to include fallback mechanisms to limit the damage that can occur as a result of such protections failing. AMI system administrators can use firmware auditing implemented as remote attestation to detect attacks and respond to them in such a way that they are prevented from inducing large-scale effects. Due to their different key storage structures, vulnerabilities in upgrade access controls and remote attestation are not likely to be closely correlated.

A desktop or mobile system can use a Trusted Platform Module (TPM) to protect and certify audit information concerning its configuration so that it can implement remote attestation [68, 58]. However, TPMs are not ideal for use in many embedded systems. TPMs impose relatively substantial cost, power, memory, and computational overheads on embedded systems. Furthermore, they provide audit data representing a limited time interval, which is incompatible with the deployment model of embedded systems such as advanced meters that operate unattended for extended periods of time.

To address the challenges put forth in this section, this dissertation makes use of integrity kernels, which are operating system kernels that provide foreign code detection or prevention services to the system. Many previous works have also described integrity kernels for various systems. Existing processor architectures for all classes of computer systems do not support integrity kernels as fully as possible. In general, it is often feasible to implement functionality in either software or hardware, with varying effects on the overall complexity of the resultant system. The specific tradeoff decisions embodied in existing processors with state-of-the-art security functionality result in integrity kernels that are unnecessarily large and complex. Such processors include at least a Memory Management Unit (MMU) that protects memory based on page tables, an IO-MMU that provides similar protections for IO memory, and often one or more hardware functions specifically intended to create and maintain isolated execution environments for integrity kernels. However, no existing processor directly detects attempts to execute unverified code, which is a core requirement for integrity kernels. This omission along with other limitations implies that all existing mechanisms exhibit deficiencies in some combination of the following requirements:

1. **Isolation**, making the integrity kernel vulnerable to compromise.

2. **Visibility**, reducing the integrity kernel's capability to detect compromises in the target system.

3. **Performance**, making integrity kernels impractical for some applications.

4. **Compatibility**, necessitating that the target be substantially modified.

1.2 Approach

1.2.1 Integrity Kernels for Commodity Microcontrollers

In this dissertation, we first describe two architectures that support remote attestation of advanced meters, which should also serve as an example of how remote attestation could be supported on other similar types of embedded systems. The first architecture is a type of integrity kernel called a *Cumulative Attestation Kernel (CAK)*, which is implemented at a low level in the meter and provides cryptographically secure audit data for an unbroken sequence of firmware revisions that have been installed on the protected system, including the current firmware. This audit data includes a cryptographic hash of the firmware. The CAK itself is never remotely upgraded, so that it can serve as a static root of trust. Our specific objective is to show that CAKs can be practically achieved on flash Microcontroller Units (MCUs). Flash MCUs are distinguished from processors in other types of computers by their onboard flash program memory, which typically contains a monolithic firmware image with a static set of applications that run in a single memory address space. In contrast, higher-end computers often run a full-featured OS such as Linux. Flash MCUs also operate at low clock frequencies and may not offer protection features such as an MMU. We account for these characteristics of flash MCUs in our design.

CAKs must provide high levels of robustness to satisfy the requirements of AMI. For example, their flash memory operations must withstand unexpected, repeated power supply interruptions. This makes CAKs resilient to battery backup failures and even permits them to operate on meters lacking battery backup. We demonstrate that CAKs are able to address these and other relevant requirements using an implementation called *Cumulative Remote Attestation of Embedded System Integrity (CRAESI)* [43]. CRAESI is targeted at a mid-range Atmel AVR32 flash MCU equipped with a Memory Protection Unit (MPU). Since the robustness requirement is unusual, we formally verify that CRAESI is resilient to unexpected, repeated power supply interruptions. This result also implies resilience to some other types of faults.

We also demonstrate the feasibility of *Cumulative Attestation Coprocessors (CACs)* for use with flash MCUs that lack an MPU and have insufficient onboard flash program memory to support a self-contained CAK. Our prototype CAC-based system is called *Cumulative Embedded System Integrity (CESIum)* and incorporates a coprocessor in addition to the main processor, both of which are 8-bit Atmel AVRs. It also uses an external flash memory. The fact that such processors lack substantial memory protection hardware greatly complicates the task of developing an integrity kernel.

We do not extensively discuss node recovery in this dissertation, since it is a distinct field of research, but we note that even recovery can be costly in AMI networks. A node's stored data may be erased during recovery, since the malicious application firmware may have corrupted the data in a way that cannot be detected after the fact. This can imply a massive loss of data on a large AMI network that could cause significant revenue loss to a business dependent on that data. By permitting individual infected nodes to be identified, and uninfected nodes to be definitively validated, cumulative attestation can minimize this revenue loss.

1.2.2 Custom Hardware to Support Integrity Kernels

We demonstrate that it is possible to dramatically reduce the complexity of an integrity kernel by adding a modest amount of hardware to a processor to directly support monitoring and isolation tasks that otherwise would have required substantial manipulations of the more general hardware protection mechanisms that are present in existing processors. The hardware extensions are sufficiently powerful that the integrity kernel built on top of them prevents foreign code from executing at all in a full-featured SPARC Linux system. One way to prevent foreign code from running on a system is to whitelist the code in legitimate applications and refuse to run any code that is not on this whitelist, thus enforcing the *eXecuting* $\rightarrow$ *Verified* property on all code. This conceptually straightforward approach has been difficult to implement in practice. It is challenging both to identify legitimate applications and to enforce the resultant whitelist. We assume that legitimate applications are known in advance and focus on enforcement in this dissertation.

Many existing integrity kernels[2] rely on virtualization implemented using an MMU. They manipulate page tables to cause a trap to the hypervisor whenever an unverified page in a Virtual Machine (VM) is about to be executed [60, 47]. Popular virtual machines provide many features, but have correspondingly large Trusted Computing Bases (TCBs) (e.g. $\sim$ 230K lines of code in Xen [10]) that may be unable to enforce isolation. Even target systems that can use minimalistic, security-oriented hypervisors may suffer from virtualization-related performance degradation. Furthermore, some virtualization approaches require substantial changes in the target system's code to make it compatible with the hypervisor.

Intel System Management Mode (SMM) can be used to overcome some of these limitations, since it provides hardware separate from the MMU to set up an isolated execution environment [10, 71]. However, the confidentiality and integrity of SMM handlers can be tricky to guarantee due to the complex interactions between the various system components involved in implementing SMM [24]. Furthermore, some system state is invisible in SMM, and SMM can only be triggered by a few types of events, such as an electrical signal at a processor pin or a write to a specific IO location. Code has been observed to execute about two orders of magnitude more slowly in SMM compared to protected mode.

We propose a new hardware mechanism that addresses the limits of other approaches. The *Integrity-Aware Processor (IAP)* is an extended SPARC processor that provides an isolated execution environment for an integrity kernel that can enforce eXecuting $\rightarrow$ Verified or provide other functionality [44]. Although the integrity kernel shares many processor resources with the target, IAP stores the integrity kernel in a completely separate address space from the target and does not permit the target to initiate any data transfers between the two spaces. On the other hand, the integrity kernel has full visibility into the target. IAP stores the entire integrity kernel on-chip to minimize access latency. Thus, integrity kernel code runs at least as fast as code in the target system, and it can be invoked with the same overhead as a native trap handler. IAP also incorporates hardware accelerators for essential cryptographic primitives.

IAP transfers control to the integrity kernel in response to several configurable conditions, including attempts by the processor to execute code that has not been verified by the integrity kernel. It includes hardware structures to track code that

[2]For the sake of consistency, we use the term "integrity kernel" to refer to all code that runs on the processor hosting the target OS and is involved in detecting and preventing the execution of foreign code, although some of that code would not ordinarily be considered part of a kernel.

has been verified and to detect attempts to modify it after it has been initially verified, subsequently causing it to be re-verified before it is re-executed. IAP monitors individual pages of memory in the virtual address space of each process, or the physical address space when the MMU is disabled. These features permit the integrity kernel to enforce eXecuting → Verified without relying on the configuration of the MMU page tables, further simplifying the TCB.

We developed the *eXecuting → Verified Enforcer (XIVE)* to demonstrate the extent to which IAP can reduce the TCB of the integrity kernel. The XIVE whitelist is located on a centralized *approver* connected to the network, to minimize the complexity of the XIVE kernel and to simplify whitelist updates. We implemented XIVE for an IAP end node based on a Field-Programmable Gate Array (FPGA), the approver on a commodity Linux host, and connected the two using 100Mbps Ethernet. The XIVE kernel contains 859 instructions, and successfully protects a Linux target system that has been slightly modified so that it is efficient to monitor and so that it tolerates sharing its physical network interface with the XIVE kernel.

Although they demonstrate some of the challenges that integrity kernels on commodity architectures must overcome, CRAESI and CESIum are not directly comparable to XIVE. CRAESI simply monitors all code that the target system executes, rather than checking it against a whitelist ahead of time. However, XIVE can be easily modified to also support this model if it is useful in a particular environment. CRAESI and CESIum also place more restrictions on the software that they support than XIVE. Some of these restrictions are imposed by the commodity flash MCUs themselves. For example, the types of MCUs targeted by CESIum do not support executing code from RAM. Although the types of MCUs targeted by CRAESI do support that, CRAESI prevents it from occurring. CESIum and CRAESI both view the entire application firmware as a monolithic image, which is consistent with the usage model of remote sensor nodes. In contrast, XIVE controls programs on a page-by-page basis and recognizes individual programs. XIVE also permits limited changes to be made to the executable pages of programs. The restrictions imposed by CRAESI and CESIum simplify their protection mechanisms, but reduce their generality and applicability to systems outside the field of remote sensor networks.

1.3 Contributions

We make the following contributions in the area of integrity kernels for commodity microcontrollers:

- We develop CRAESI, the first technology to support remote attestation for mid-range flash MCUs in remote sensor networks, which are subject to high jitter and permit sensors to collude with external computers. CRAESI's threat model admits malicious application firmware. We implemented it as a CAK that leverages the moderately-sized flash memory and MPU of an Atmel AVR32 MCU without requiring a coprocessor.

- We present a formal proof that CRAESI has certain security and fault-tolerance properties. We accomplish this using the Maude model checker. This is the first formal proof that a system is tolerant to power supply interruptions.

- We develop CESIum, the only approach to remote attestation for remotely-upgradeable, low-end flash MCUs in remote sensor networks that is invulnerable to control-flow attacks launched by malicious application firmware. It incorporates a security coprocessor and an external flash memory.

We make the following contributions in the area of integrity kernels for an extended processor architecture:

- We develop IAP, the only processor architecture with hardware support for directly detecting attempts to execute unverified code.

- We develop the XIVE kernel for IAP, the most compact integrity kernel that is capable of enforcing eXecuting $\rightarrow$ Verified.

1.4 Dissertation Scope and Organization

We investigate *compact integrity-aware architectures* in this dissertation. That investigation involves hardware, software, and firmware components, as well as formal proofs. The size of a system can be measured in several ways. Like many other works, this dissertation primarily uses source lines of code as the metric. In general, a compact system can be expected to exhibit fewer security vulnerabilities than a larger system, assuming that the two systems have similar characteristics such as their implementation and verification techniques, their overall purpose, and their history of usage and testing [4]. Furthermore, compact systems are generally more amenable to formal verification [7]. Additionally, binary compactness is important for integrity kernels on embedded systems, because those systems have severe resource limitations. Our thesis is:

It is possible to develop compact integrity kernels to protect commodity microcontrollers in remote sensor networks, and a custom hardware architecture can enable the construction of compact integrity kernels with superior isolation, visibility, performance, and compatibility.

Chapter 2 contains background material that is helpful in understanding the rest of the dissertation. It contains information on remote attestation, AMI, formal verification, and processor technologies.

Chapter 3 discusses integrity kernels for commodity microcontrollers. The CRAESI CAK for the Atmel AT32UC3A0512 AVR32 microcontroller is implemented using C++ and is situated within a prototype that emulates an advanced electric meter. We use the Maude model checker to formally verify that CRAESI correctly implements core security functions related to audit log maintenance and that it properly updates its audit log and internal filesystem in the presence of repeated, unexpected power supply interruptions, provided that the power supply is eventually restored for a sufficient period of time. The CESIum CAC-based system uses an Atmel ATmega644V AVR microcontroller as a security coprocessor that monitors the main microcontroller, an Atmel ATmega2560 AVR. The firmware for both microcontrollers is implemented using C and assembly language. The prototype also incorporates an external flash memory.

Chapter 4 presents our extended processor architecture, IAP, and discusses integrity kernels for that architecture. IAP uses the LEON3 SPARC soft core as the basis for a prototype. It is synthesized from VHDL for use on an FPGA. The XIVE integrity kernel is written entirely in SPARC assembly language and is more isolated, comprehensive, and compatible than other integrity kernels that are capable of enforcing eXecuting → Verified. XIVE monitors all of the code that is executed by a Linux target system running on IAP. We evaluate the performance of a prototype system based on the XIVE kernel and a network-based whitelist server we developed using multi-threaded C++ on Linux and show that it is adequate. We also show that IAP imposes less overhead on integrity kernels compared to other processors.

Chapter 5 compares our approaches to those in related work. It covers works that rely on coprocessors, software, hypervisors, and processor extensions to implement integrity kernels.

Chapter 6 summarizes our contributions and proposes future work.

Chapter 2

Background

This chapter discusses background material that is helpful for understanding the rest of this dissertation. It briefly explains remote attestation. Next, it describes the nature, scale, and motivations of ongoing AMI deployments, as well as anticipated benefits from AMI. Then it provides a brief introduction to formal verification using Maude. Finally, it discusses relevant processor technologies and the memory management and protection mechanisms that they provide.[1]

2.1 Remote Attestation

Remote attestation is a process whereby a remote verifier $\mathcal{V}$ can obtain certified measurements of parts of the state of a system C. A variety of protocols can be used to accomplish this. They usually involve at least two messages. The request $\mathcal{V} \xrightarrow{\langle v \rangle} C$ contains a nonce v used to ensure the freshness of the attestation results. The response $C \xrightarrow{[\langle v, \sigma \rangle]_{RTMC}} \mathcal{V}$ is digitally signed by the Root-of-Trust for Measurement (RTM) of C (RTMC) to certify that it has not been tampered, and contains the nonce v as well as a record of the system's state σ. Of course, this assumes that the system contains some RTMC that is capable of securely recording and certifying the system's state. On desktop PCs, the TPM and supporting components in the system software often fulfill this role.

A TPM is typically a hardware security coprocessor that comprises several internal peripherals coordinated by a central microcontroller core [68]. It is intended to be difficult to remove from the platform in which it was originally installed. It is also designed to make physical tampering evident upon subsequent physical inspection. The TPM contains several keypairs. Two of them can be used to digitally sign internal registers that contain cryptographic hashes. These registers

[1]This chapter includes material from a previous publication by LeMay, Gross, Gunter, and Garg [42]. It also includes material from a previous publication by LeMay and Gunter [43].

are called Platform Configuration Registers (PCRs). The TPM implements an "Extend" function that requires a hash value as a parameter and then updates the hash value in a particular PCR by appending the new hash to the old PCR value, hashing the result, and storing it in the PCR. The OS on C maintains a log of information that can be used to evaluate its configuration and state and performs an extend operation to commit each new log entry as it is added. To generate a trustworthy attestation using the basic protocol described above, the main processor on C must send the nonce to the TPM and request that it digitally sign ("Quote") the PCRs. The processor is assumed to not have the capability to forge these signatures, since the TPM's private keys are never released by the TPM unless it is physically compromised. Therefore, for the protocol to proceed, the TPM must return the signed attestation data to the processor, and the processor must then return that signature along with the log that is required to interpret the attestation to $\mathcal{V}$. The TPM contains many other structures to support true random number generation, cryptography, and other functions.

Well-designed systems include multiple layers of protections to prevent compromises, including access controls for installation privileges and signatures on code that are verified before installation. Remote attestation provides an additional layer of defense in the event that these protections fail, hence providing defense-in-depth. This is similar to the synergistic relationship between Network Intrusion Detection Systems (NIDSs) and firewalls. A NIDS detects attacks that bypass firewalls, leading to faster attack recovery and a subsequent strengthening of firewall rules. These mechanisms work well together because of their distinct failure modes. Typical upgrade controls that require firmware to be signed can be compromised if the private keys used to sign firmware are compromised, or the upgrade controls are bypassed by a buffer overflow or other type of attack. Modern embedded systems can run complex software stacks that may be vulnerable to attacks similar to those that have plagued server and desktop machines. Even if a different key is used to sign firmware upgrades for each node on a network, those private keys are all likely to be stored in a central repository. The compromise of the repository could lead to the compromise of all systems on that network. In contrast, CAKs and CACs store their private keys within individual meters that are geographically scattered, greatly increasing the cost of compromising large numbers of private keys. Only the availability and authenticity of the corresponding public keys must be ensured to provide secure auditing capabilities. This is generally a more tractable problem than ensuring long-term confidentiality of a centralized private key repository.

Even if a system is never compromised, remote attestation is useful when multiple parties are authorized to upgrade or use the system and they must be able to verify that the configuration changes made by other parties are acceptable. For example, government regulators could query an advanced meter to obtain an unforgeable guarantee that it is using firmware that provides accurate meter readings.

2.2 Advanced Metering Infrastructure

Advanced electric meters are embedded systems deployed by utilities in homes or businesses to record and transmit information about electricity extracted from the power distribution network and potentially to support more advanced functionality. They arose out of automated meter reading, which simply involves remote collection of meter data. However, AMI can support new applications based on bidirectional communication, such as the ability to manipulate power consumption at a facility by sending a price signal or direct command to its meter (demand response). AMI networks are being deployed on a massive scale [5]. A report by Pike Research states that more than 250 million meters will be deployed worldwide by the year 2015 [57]. AMI is a particularly good example of a remote sensor network and a good benchmark for study because of its nascent but real deployment and rich set of requirements.

The sophisticated functionality of advanced meters creates numerous attack scenarios and increases the likelihood that they will contain security vulnerabilities linked to firmware bugs. An outage of the meters in a region could entail a huge financial loss for a utility. The “Guidelines for Smart Grid Cyber Security” published by NIST specifically call for remote attestation of smart grid components [1]. In a previous work we further motivated the use of attestation to provide AMI security [42].

Other embedded systems could also benefit from CAK-supported intrusion detection. Intelligent Electronic Devices (IEDs) used in electrical substations to monitor and control the transmission and distribution of electricity often support remote firmware upgrades and can exert more direct control over the flow of electricity than demand-responsive meters [29].

We now provide additional details on AMI. In the future, it will afford a number of potential advantages to energy service providers, their customers, and many other entities [28]:

1. **Customer control:** Customers gain access to information on their current energy usage and real-time electricity prices.

2. **Demand response:** Power utilities can more effectively send control signals to advanced metering systems to curtail customer loads, either directly or in cooperation with the customer's building automation system.

3. **Improved reliability:** More agile demand response can improve the reliability of the distribution grid by preventing line congestion and generation overloads. These improvements could also reduce the strain on the transmission grid.

There are several distinct categories of advanced metering systems that support the functionality discussed above with varying degrees of success. The least capable systems use short-range radio networks and may be less expensive to deploy initially, but they require readers to drive by in vans to read the meters. More capable systems support unidirectional network communication from the meter data management service, and the most capable systems have fully bidirectional network connections with the meter data management service. We focus on meters with bidirectional connections in this dissertation. AMI networks with connectivity to the meter data management service can distribute real-time pricing schedules to meters, which can influence customer behavior and induce manual or automatic demand response actions [15]. They can also support direct control signals.

In Figure 2.1, we show how a full-featured bidirectional metering network could be organized. The network is divided into two main domains that are connected via a Field-Area Network (FAN). The first domain houses the meter data management service and the energy service provider that controls the physical delivery of electricity. The second domain comprises the metered premises, which may have mesh network connections between themselves to extend the overall reach of the AMI network. Each of these premises may also be equipped with a Home-Area Network (HAN) containing an in-home display, which interacts with the meter and intelligent appliances and perhaps a home energy dashboard that provides complementary features to those of the in-home display. We have investigated the challenges involved in coordinating such a complex network and proposed a hierarchical approach that accounts for the various levels of functionality present in various devices within the HAN [45].

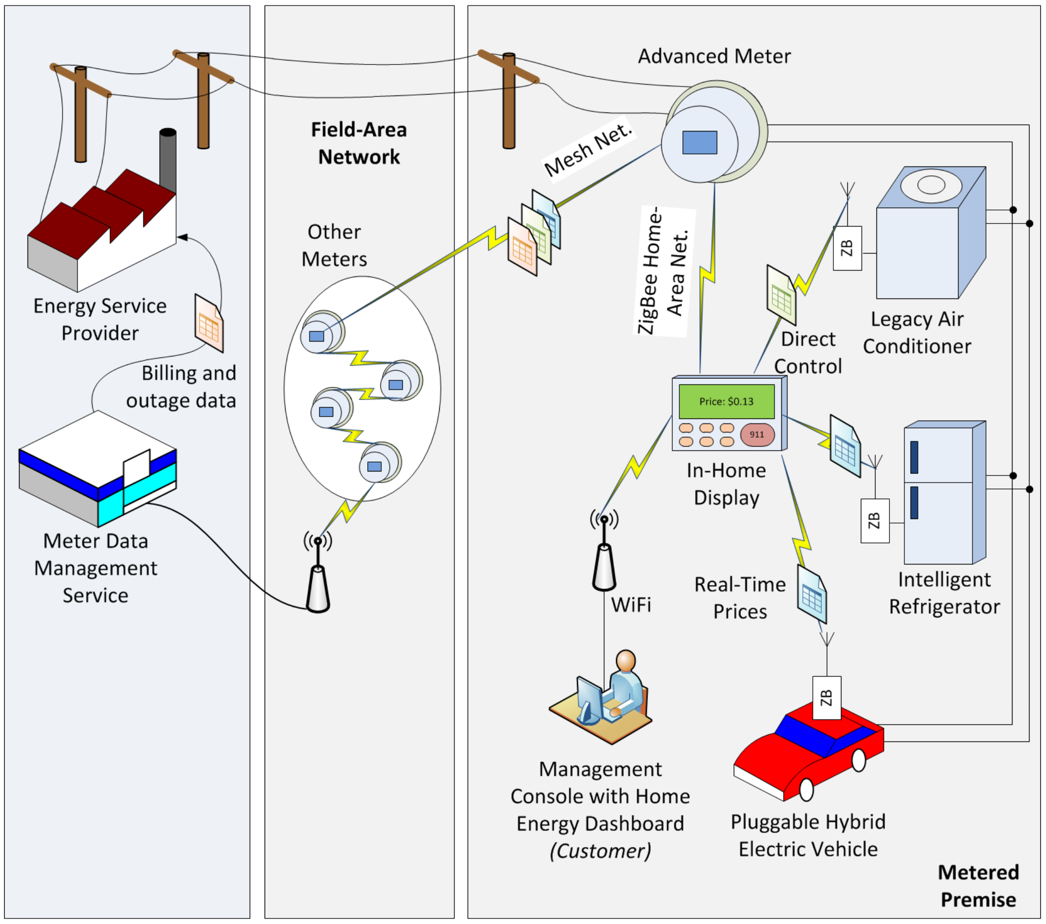


Figure 2.1: Full-featured bidirectional metering network interactions.

Operator	Name	Description
$\bigcirc\varphi$	Next	φ holds in the next state.
$\varphi\ \mathcal{U}\ \psi$	Until	ψ holds in the current or a future state, and φ holds until that state is reached.
$\Diamond\varphi$	Eventually	φ holds in some subsequent state.
$\Box\phi$	Henceforth	ϕ must hold in all subsequent states.
$\varphi\ \mathcal{W}\ \psi$	Unless	φ holds in all states until ψ holds, or forever if ψ is never satisfied.
$\varphi \Rightarrow \psi$	Strong Implication	ψ holds in any state in which φ is satisfied.

Table 2.1: Temporal modal operators used in LTL formulas.

2.3 Formal Methods

Formal methods are used to verify correctness and fault-tolerance properties of CRAESI in §3.4. Specifically, model checking is a methodology for systematically exploring the entire state space of a model and verifying that specific properties hold over that entire space. Maude is the name of a language as well as a corresponding tool that supports model checking based on rewriting logic models and Linear Temporal Logic (LTL) properties [21]. Essentially, rewriting logic provides a convenient technique to express non-deterministic finite automata. Maude is a multi-paradigm language, and supports membership equational logic, rewriting logic, and even has a built-in object-oriented layer. We use Maude for our verification tasks.

Maude provides a `search` function that can be used to explore all distinct states that can be reached from an initial state. The search command can be parameterized to only display states that satisfy a particular property, and this can be used to perform basic model checking. This is only suitable when the desired state can be identified by a simple set of propositions combined using logical connectives ($\neg, \wedge, \vee, \rightarrow$) on that state, not considering any preceding states. Appropriate propositions must be defined for the particular model under consideration.

For more sophisticated model-checking operations, theorems and lemmata can be formalized using LTL. An LTL formula is a predicate over a sequence of states. Each formula comprises propositions that are connected with logical connectives and the temporal modal operators described in Table 2.1.

2.4 Processor Technologies

We deal with several processor technologies in this dissertation, with a special emphasis on flash MCUs and FPGAs. We describe the relevant characteristics of these technologies in this section and compare them against others.

The flash MCUs in this dissertation are commodity processors that incorporate a microprocessor core, data RAM, flash program memory, EEPROM persistent data memory, clock generation circuitry, and peripherals such as various serial and memory interfaces, analog-to-digital converters, etc. They are essentially complete computers in a single chip, lacking only a power supply and connections to devices that enable them to meaningfully interact with the physical world. Another way to think of a flash MCU is as a single chip that contains analogous components for most of the items found inside the case of a typical desktop PC. A PC requires connections to a monitor, keyboard, power cord, and perhaps other devices to do useful work, just as flash MCUs are reliant on external components to interact with the physical world. However, flash MCUs have very limited capabilities and low levels of performance, because they target applications with tight energy and cost constraints. A wide variety of flash MCUs are currently available, based on 8-bit through 32-bit instruction set architectures. The security features in these designs also vary widely, with some providing simple support for isolating a bootloader from an application and others featuring MPUs that support separate privilege levels and fine-grained memory access controls.

Most processors today, including flash MCUs, are manufactured as Application Specific Integrated Circuits (ASICs), which means that their designs are etched into silicon in such a way that they implement a single circuit design that provides processor functionality. An alternative approach to implementing processors uses FPGAs, which are chips that can be reconfigured many times to implement different circuits. ASICs have many advantages over FPGAs that justify their popularity in spite of their inflexibility. For example, an FPGA consumes more area, energy, and time to perform a computation compared to an ASIC. However, FPGAs are very useful for performing processor design research and can even be used in certain applications that are not sensitive to the drawbacks of FPGAs. FPGAs are also being combined with ASICs in some cases to provide benefits from both processor technologies.

FPGAs must be configured to implement a particular circuit using a bit file, which can be generated from a Hardware Description Language (HDL). Verilog and VHDL are popular HDLs. A hardware design encoded in an HDL can represent the hardware's circuit at various levels of abstraction. It resembles the source code of a software program in many ways, and can even contain sequential statements similar to those in source code from an imperative software programming language. However, the bit file is not analogous to a binary program comprising a sequence of instructions. The HDL source code undergoes a multi-step process that ultimately transforms it into the bit file, which represents a low-level configuration of the individual circuit elements within the FPGA that will then implement the desired overall circuit. When the bit file contains the configuration information to implement an entire processor within the FPGA, we refer to that processor as a "soft core."

2.5 Memory Management and Protection

Some of the processors in this dissertation primarily rely on an MPU or MMU to protect and manage memory accesses from software. This section describes the basic concepts underlying these mechanisms as they are implemented in the specific processors in this dissertation.

An MPU is simpler than an MMU and simply defines and controls access to regions of memory. Note that it does not perform any mapping of memory addresses. The OS configures the MPU to divide the processor's physical address space into various regions, and to control access to those regions according to the current mode of the processor when it is executing software. For example, the software may be allowed to read, write, and execute some region of memory when it is executing in supervisor mode, as would be the case when the operating system kernel is active. It could be restricted so that it is only able to read that region when it is executing in user mode.

An MMU maps addresses in a virtual address space to physical addresses that correspond to physical memory locations and also enforces access controls during that process. These mappings are defined using page tables. Page tables are arranged hierarchically and terminate in page table entries that either reference a specific page of physical memory or are invalid, indicating an undefined region of virtual addresses. To map a virtual address to a physical one, the processor performs an iterative process to walk through these page tables until it reaches a page table entry. At each step, it uses a specific portion of the original virtual address to perform the lookup to identify either the page table entry or the pointer to the next page table. Once a valid page table entry has been accessed, the processor combines it with the remaining portion of the original virtual address and uses that as the physical address. Walking through page tables is an expensive process, so the processor caches recently-accessed page table entries in structures called Translation Lookaside Buffers (TLBs). If the page table walker encounters an invalid page table entry or one that does not permit the requested type of access, it generates a trap to the OS.

Operating systems typically allocate a separate virtual address space for each process. Thus, it is not sufficient for the various caches in the processor (including TLBs) to simply label entries with virtual addresses, since multiple processes may use identical virtual addresses to refer to different physical addresses. The processes may also be assigned different permissions for the same address. To deal with this issue, the processor associates a unique context value with each process, and includes that context value in each cache entry that is created when the process is active.

Chapter 3

Commodity Cumulative Attestation Kernels and Coprocessors

We demonstrate in this chapter that we can develop compact integrity kernels for commodity flash MCUs, and that those integrity kernels exhibit good performance compared to well-known alternative approaches. We accomplish this by first presenting a threat model and a set of requirements in §3.1 for these integrity kernels. Second, we propose a design that satisfies those requirements on a flash MCU with a modest amount of flash memory and an MPU in §3.2. We present results from an experimental evaluation of a prototype implementation of that design, called CRAESI, in §3.3. We then formally analyze CRAESI with respect to important fault-tolerance and correctness properties in §3.4, using the Maude model checker.

CRAESI is not applicable to flash MCUs that have a small amount of flash memory or lack an MPU. However, such MCUs are still used in security-critical environments. Thus, we discuss how the design of CRAESI can be adapted to support such MCUs in §3.6. We also evaluate the performance of the adapted design on a prototype system called CESIum.[1]

3.1 Threat Model and Requirements

3.1.1 Threat Model

Data integrity on meters can be compromised by malicious application firmware in various ways, as shown in Figure 3.1. Actuator controls can also be abused. A typical remote attestation scheme provides evidence of the integrity of data (such as firmware) at the time an attestation report is requested. Such a system is vulnerable to what one might call Time-Of-Use-To-Time-Of-Check (TOUTTOC) inconsistencies (dual to the more familiar Time-Of-Check-To-Time-Of-Use (TOCTTOU)

[1]This chapter includes material from a previous publication by LeMay and Gunter [43].

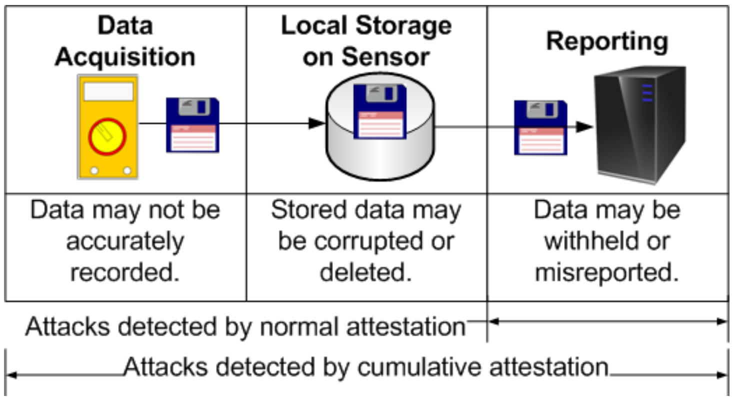

Data Acquisition
Local Storage on Sensor
Reporting
Data may not be accurately recorded.
Stored data may be corrupted or deleted.
Data may be withheld or misreported.
Attacks detected by normal attestation
Attacks detected by cumulative attestation

Figure 3.1: Three modes of attack on sensor data available to malicious application firmware running during various lifetime phases occupied by that data.

inconsistencies) wherein data was inaccurately recorded, corrupted, or deleted, or actuator controls were abused, before the time of attestation if the system was subsequently reset. In contrast, cumulative attestation detects such attacks.

We assume that an attacker is capable of communicating with a protected system over a network and installing malicious application firmware. We also assume that the attacker has a priori knowledge of the layout of the kernel's code and data memory spaces, as well as their static contents, but not the contents of dynamic variables and static values that vary between kernels.

"Ordinary" environmental phenomena must not cause any of the security requirements of the kernel to be violated. An example is an accidental power supply interruption, unless the system has a robust, trusted power supply. On the other hand, a bit flip caused by cosmic radiation would be considered an extraordinary phenomenon in most ground-based embedded systems. These examples make it clear that the definitions of ordinary and extraordinary will vary based on a system's intrinsic characteristics and its environment. In this paper, we only include accidental power supply interruptions in our threat model, although we discuss other types of faults that have similar effects and are therefore handled by the same fault-tolerance mechanisms. We also exclude physical attacks on microcontrollers such as fault analysis, silicon modifications, and probing [6, 31]. The use of a CAK does not exclude tamper-resistance, but CAKs address remote attacks rather than local, physical attacks which are generally much more expensive than remote attacks. Large-scale remote attacks potentially enable different classes of attack outcomes (such as blackouts in our example).

The security of remote attestation based on a CAK depends upon the fact that application firmware runs at a lower privilege level than the CAK and is not permitted to access security-critical memory and peripherals. This excludes a wide variety of attacks, such as Cloaker [22]. The specific peripherals that are considered security-critical will vary between microcontrollers.

Note that a CAK does not detect attacks that succeed by simply modifying data RAM. Although data RAM is not executable by the application, corruption in data RAM can lead to system compromises in other ways [34]. However, it is prohibitively expensive to record changes to data RAM. It is also more challenging to characterize all legitimate values of data RAM. This limitation implies that return-oriented programming can potentially be used to corrupt the control flow of an audited firmware image to implement an attack. However, it can be more difficult to construct a return-oriented program than it is to construct a program intended for native execution. For example, the targeted firmware must contain a sufficient set of "gadgets" to implement the desired attack [30].

3.1.2 Requirements

The basic security and functional requirements for a CAK are that it maintain an audit log of application firmware revisions installed on a meter, and that it make a certified copy of that log available to authorized remote parties that request it. It must satisfy the following properties to provide security: *1) Comprehensiveness:* The audit log must represent all application firmware revisions that were ever active on the system. Application firmware is considered to be active whenever the processor's program counter falls somewhere within its installed code space. *2) Accuracy:* Whenever application firmware is active, the latest entry in the audit log must correspond to that firmware. The earlier entries must be chronologically ordered according to the activation of the firmware revisions they represent.

We define the following requirements for a CAK based on the characteristics and constraints of advanced meters:

1. **Cost-effectiveness:** Even the smallest added expenses in advanced meters become significant when multiplied for massive deployments.

2. **Energy-efficiency:** Some embedded systems are critically constrained by limited energy supplies, often provided by batteries. Although meters are attached to mains power, they may be constrained to low energy consumption to reduce energy costs.

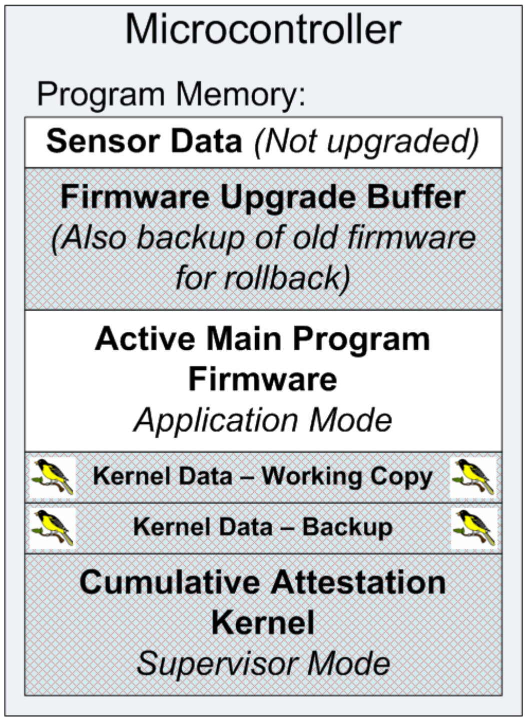

Microcontroller
Program Memory:
Sensor Data (Not upgraded)
Firmware Upgrade Buffer
(Also backup of old firmware for rollback)
Active Main Program Firmware
Application Mode
Kernel Data – Working Copy
Kernel Data – Backup
Cumulative Attestation Kernel
Supervisor Mode

Figure 3.2: The general CAK program memory layout. The birds represent "canary" values.

3. **Suitability for hardware protections:** The CAK must be adapted to the protection mechanisms provided by the processor on which it runs.

3.2 CAK Design

We now present a kernel design that satisfies the requirements. The basic flash memory layout of the system is depicted in Figure 3.2. The executable code for the CAK is located at the beginning of memory, where bootloaders are usually stored. Above that, two redundant regions are used to store data used by the kernel. The *Installed Region* is the only memory containing instructions that can be executed in user mode. The *Upgrade Region* is used to buffer firmware upgrades. Finally, *Sensor Data* can potentially be used by the application to store arbitrary non-executable data.

The content of each *Kernel Data* section is divided into several regions, and contains the following:

3.2.1 Audit Log

Each audit log entry is denoted as $\epsilon_i = \langle \tau, \eta, \theta \rangle$, where $\tau \in \{hash, chain\}$ specifies the type of the entry, η specifies the event that caused ϵ to be recorded in the log if it was not recorded as a result of a successful upgrade, and θ is a hash value $h(\mathcal{F}_i)$ if $\tau = hash$ and $\mathcal{F}_i$ is the currently installed firmware image, or a hash chain if $\tau = chain$. The audit log $AL = (\epsilon_0, \epsilon_1, \ldots \epsilon_{n-1})$ when $|AL| = n$ such that $\mathcal{F}_i$ was installed immediately after $\mathcal{F}_{i-1}$. It is possible for AL to overflow memory, so it can be divided into two lists $AL_{ovf} = (\epsilon_0, \epsilon_1, \ldots \epsilon_m)$ and $AL_{recent} = (\epsilon_{m+1}, \epsilon_{m+2}, \ldots \epsilon_{n-1})$. The maximum length of AL_{recent} is dictated by the capacity of the flash. When it overflows, the entry $\epsilon_{chain} = \langle chain, none, h(h(\ldots \| \epsilon_{m-1}) \| \epsilon_m) \rangle$ is included in the audit log memory region, where $\|$ is used to indicate concatenation. Its hash value represents a left fold of AL_{ovf} with the function $h_{fold}(x, y) = h(x \| y)$. $AL_{inmem} = AL$ when $|AL_{ovf}| = 0$ and $AL_{inmem} = (\epsilon_{chain}) + AL_{recent}$ otherwise. A counter $\lambda = |AL|$ is also included.

If AL has overflowed, the remote party performing the attestation must already know $(h(\mathcal{F}_0), h(\mathcal{F}_1), \ldots h(\mathcal{F}_m))$. This is a reasonable assumption if the embedded system is used by a group of remote parties that can communicate with all parties that have installed new firmware revisions on the system. In that case, the party verifying the attestation can request that the updaters provide whatever hash values the verifier does not yet know. It can then verify the current hash chain.

3.2.2 Asymmetric Keypairs

The public and private keys for keypair P_x are denoted as Y_x and R_x, respectively. $R_{\mathcal{F}}$ is used to sign the firmware audit log during attestation operations. P_{DH} is used during Diffie-Hellman key exchanges. R_Ω is used to sign $Y_{\mathcal{F}}$ and Y_{DH}. It is generated by the CAK using its built-in random number generator when it is first started and stored in memory, or burned into fuses at the factory in such a way that no entity, including the manufacturer, can determine its value. Counters $\lambda_{\{\mathcal{F}, DH, \Omega\}}$ are used to record the number of signatures generated by the corresponding private keys. $P_{\mathcal{F}}$ and P_{DH} will be individually refreshed when their associated counters reach a threshold value.

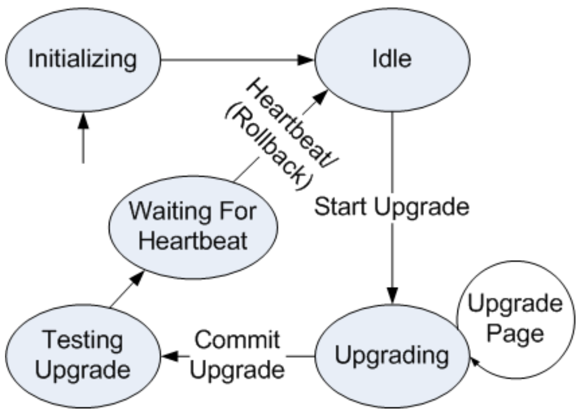


Figure 3.3: A basic state machine representation of CAK operation, in which transitions are generated by the specified commands.

3.2.3 System State

An explicit state variable σ is used to control transactions. States in the automata in Figure 3.3 illustrate the possible values of σ.

The CAK satisfies the Comprehensiveness and Accuracy requirements by controlling all access to the low-level firmware modification mechanisms in the system. The state machine in Figure 3.3 manages the application firmware upgrade process within the CAK. The transition labels not in parentheses are commands that can be issued by the application to cause itself to be upgraded. The current state is recorded in σ. The "Waiting for Heartbeat" state causes the application firmware to be reverted to its previous revision if no heartbeat command is received within a certain period of time. Any unexpected command received by the CAK will be ignored.

Three additional commands not shown in the figure can be executed by an application to: *1) Quote:* digitally sign and transmit a copy of AL_{inmem}, including a nonce for freshness, *2) Retrieve Public Keys:* retrieve $P_{\mathcal{F}}, P_{DH}, P_{\Omega}$ signed using R_{Ω}, and *3) Handshake:* perform a Diffie-Hellman key exchange. The Handshake command demonstrates how the asymmetric cryptography implemented within the kernel can be used to perform operations directly useful to the application (establish a symmetric key with a remote entity, in this case), to defray the memory space that the CAK requires. More general access could be provided in future designs, but would complicate the security analysis of the API.

Transactional semantics must be provided for all the persistent data used by the kernel. This design accomplishes that by maintaining redundant copies of all persistent data in a static “filesystem” $FS = \langle \gamma_0, \Phi_w, \gamma_1, \gamma_2, \Phi_p, \gamma_3 \rangle$ where each γ_i ($i \in \{0, 1, 2, 3\}$) is a Boolean “canary” flag, and Φ_w and Φ_p are tuples of the form $\langle \sigma, P_{\mathcal{F}}, \lambda_{\mathcal{F}}, P_{DH}, \lambda_{DH}, P_\Omega, \lambda_\Omega, \sigma_{upg}, AL_{inmem} \rangle$, where σ_{upg} encodes the state of the upgrade process, as explained below. The tuple Φ_w is a working copy that is modified by the kernel and Φ_p is a persistent copy that provides redundancy. The working copy update process is described in Listing 1.

It is more conventional to represent a filesystem as a relation between filenames and data, and in fact we use that representation in our formal analysis of this filesystem’s fault tolerance in §3.4. In that case, $FS \subset FN \times FD$ where FN is the set of filenames and FD is the set of all possible file data values.

Listing 1 Update filesystem working copy.

```
procedure FsStore(addr, data)
    $\gamma_0 \leftarrow False$
    $\gamma_1 \leftarrow False$
    $\Phi_w \leftarrow \textsc{Insert}(addr, data, \Phi_w)$
    $\gamma_0 \leftarrow True$
    $\gamma_1 \leftarrow True$
end procedure
```

The copies of the filesystem have canary values γ_i before and after the file data as depicted in Figure 3.2. Whenever a file in the working copy is modified, canaries γ_0 and γ_1 are first invalidated and then re-initialized after the file has been written. An unlimited number of modifications can be made to the working copy within a single transaction. When the transaction is finally committed, γ_2 and γ_3 are first invalidated. Next, $\langle \gamma_0, \Phi_w, \gamma_1 \rangle$ is copied over $\langle \gamma_2, \Phi_p, \gamma_3 \rangle$. If this copy operation completes successfully, canaries γ_2 and γ_3 will be automatically restored (Listing 2).

Listing 2 Update filesystem persistent copy.

```
procedure FsCommit
    $\gamma_2 \leftarrow False$                    ▷ $aborted(PostCritical)$ $(\neg aborted(PreCritical))$
    $\gamma_3 \leftarrow False$
    $\langle \gamma_2, \Phi_p, \gamma_3 \rangle \leftarrow \langle \gamma_0, \Phi_w, \gamma_1 \rangle$
end procedure
```

The presence of comments in the pseudocode, like “*aborted*(*PostCritical*) (¬*aborted*(*PreCritical*))” on the right side of the first line indicates that the referenced propositions hold after that line has completed its execution. These propositions are described in §3.4.

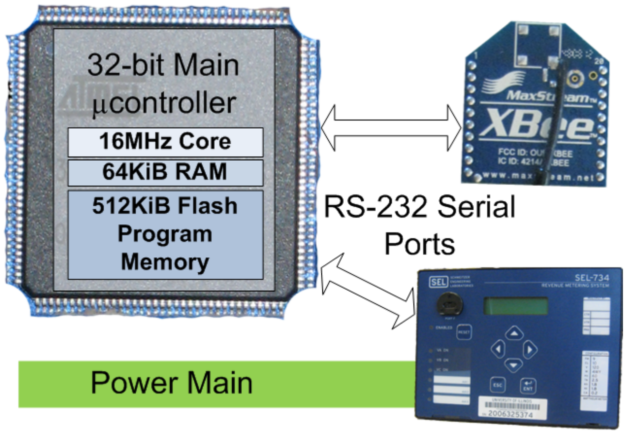


Figure 3.4: Prototype hardware components and interconnects.

When the processor boots up, it initializes the filesystem, which involves checking the canaries (Listing 3). At most one copy will have invalid canaries, and the other copy would then be used to restore the invalid copy. If both sets of canaries are valid, but the filesystem data is not identical, the persistent copy will be used to restore the working copy.

Listing 3 Initialize CAK filesystem.

```
procedure FsInit
    if $\langle \gamma_2, \gamma_3 \rangle \neq \langle True, True \rangle$ then
        $\Phi_p \leftarrow \Phi_w$
    else if $\Phi_w \neq \Phi_p$ then
        $\Phi_w \leftarrow \Phi_p$
    end if
end procedure                                    ▷ sff-inited
```

The application firmware upgrade process is also fault-tolerant, but has significantly different fault-tolerance semantics than the filesystem. Two firmware regions are maintained in the system's flash. The upgrade region is used to store a firmware upgrade as it is uploaded. The installed region is the region actually executed when the application firmware is active. The commit process sequentially swaps pages in the two regions, using a page-sized staging area elsewhere in kernel program memory (Listing 4). The data in the two regions has been completely swapped at the end of the commit process. A status value σ_{upg} is stored in the filesystem and updated as the commit process progresses to enable recovery after a power failure that interrupts the process.

Every time the meter boots, the processor immediately transfers control to

Listing 4 Load firmware upgrade into executable space.

```
procedure UpgradeCommit
    while $\sigma_{upg}.n < pagecnt$ do    ▷ $n$ is initialized to 0 when an upgrade is first initiated, and is not reinitialized here, because it is used to recover from unexpected interruptions in the upgrade process.
        if $\sigma_{upg}.stage = Staging$ then    ▷ stage is initialized to Staging when an upgrade is first initiated.
            $codeStagingArea \leftarrow upgradeRegion_n$
            $\sigma_{upg}.stage \leftarrow BackingUp$
            FsCommit
        else if $\sigma_{upg}.stage = BackingUp$ then
            $upgradeRegion_n \leftarrow installedRegion_n$
            $\sigma_{upg}.stage \leftarrow Finishing$
            FsCommit
        else if $\sigma_{upg}.stage = Finishing$ then
            $installedRegion_n \leftarrow codeStagingArea$
            $\sigma_{upg}.n \leftarrow \sigma_{upg}.n + 1$
            $\sigma_{upg}.stage \leftarrow Staging$
            FsCommit
        end if
    end while
    $\sigma \leftarrow TestingUpgrade$
    FsCommit
end procedure
```

the DirInit procedure in the CAK (Listing 5). The CAK first initializes the memory protections, performs filesystem recovery if necessary, and completes the application firmware upgrade transaction if one was interrupted by a power failure. It then generates a cryptographic hash of the firmware and compares it to the latest audit log entry. If they differ, it extends the log with a new entry. Finally, it transfers control to the application.

Listing 5 Initialize director upon system reset.

```
procedure DirInit
    FsInit
    if IsUpgrading then
        UpgradeCommit
    end if
    if $\sigma = Init \lor \sigma = Idle$ then
        DirPrep($None, Idle$)
    else if $\sigma = Upgrading$ then
        DirPrep($UpgradeAborted, Idle$)
    else if $\sigma = TestingUpgrade$ then
        DirPrep($None, WaitingForHB$)
    else if $\sigma = WaitingForHB$ then
        DirPrep($UpgradeHBFailed, Idle$)
    end if
end procedure
procedure DirPrep($\eta, \sigma^+$)
    $\sigma \leftarrow \sigma^+$
    FsCommit
    if $\eta \neq None \lor |AL_{\text{inmem}}| = 0 \lor \epsilon_{n-1} \neq \langle \eta, h(\mathcal{F}_n) \rangle$ then
        LogExtend($\langle \eta, h(\mathcal{F}_n) \rangle$)
        FsCommit
    end if
    JumpMain                                        ▷ appfw-active
end procedure
```

Both fault-tolerance processes are analyzed in §3.4 to ensure that the particular memory manipulations they use correctly recover from accidental power supply interruptions.

3.3 CAK Implementation and Evaluation

In this section we present CRAESI, a prototype standalone CAK. The purpose of this prototype is to demonstrate that our design satisfies the practical requirements put forth in §3.2, and to obtain preliminary performance, cost, and power-consumption measurements. However, these preliminary measurements do not indicate the parameters that will be exhibited by commercial implementations, since our prototype relies heavily on unoptimized software.

3.3.1 Hardware Components

Our prototype implementation comprises five distinct devices. The first is an Atmel ATSTK600 development kit containing an AT32UC3A0512 AVR32 microcontroller with a 3.3V supply voltage. The second device is a Schweitzer Engineering Laboratories SEL-734 substation electrical meter. The SEL-734 has a convenient RS-232 Modbus data interface. We could have used any similar device in our experiments since it simply serves as a realistic data source connected to the AVR32 microcontroller. Third, we use a standard desktop PC to communicate with the AVR32 microcontroller over an RS-232 serial port from a Java application that issues Modbus commands. The final two devices are paired ZigBee radios that relay RS-232 data between the PC and AVR32 microcontroller.

3.3.2 Application Firmware

We prepared two application firmware images for our experiments. They both implement Modbus master and slave interfaces, where the master communicates with the meter over an RS-232 serial port, and the slave accepts commands from the PC over the ZigBee link and either passes them to the kernel or handles them directly if they are requesting data from the meter. The first image accurately relays meter data, whereas the second halves all meter readings, as might be the case with a malicious firmware image installed on an advanced meter by an unethical customer.

CRAESI would interfere with the operation of existing embedded operating systems that require access to security-critical peripherals and memory areas. However, virtualization techniques could be used to accommodate those accesses, given sufficient resources to implement the virtualization.

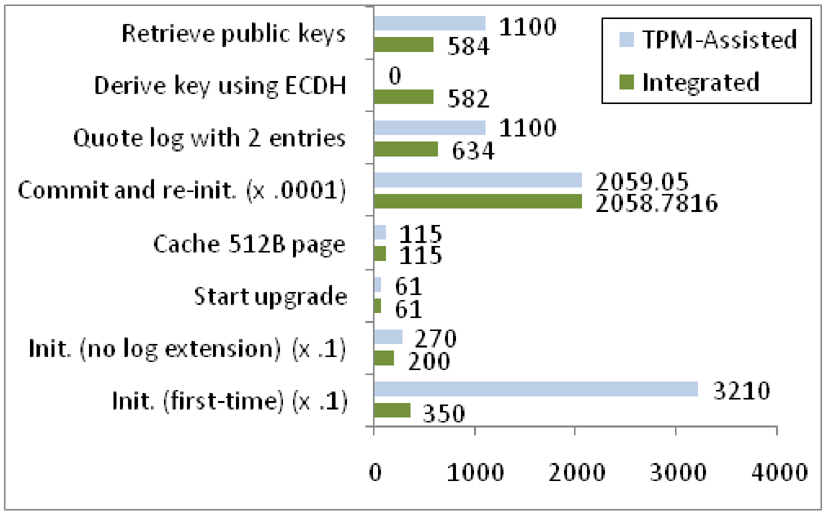
Module	Lines of Code
Core	810
Crypto	5684
Filesystem	160
Hardware Management	256
TOTAL	6910

Table 3.1: Lines of C++ code in each CRAESI kernel component.

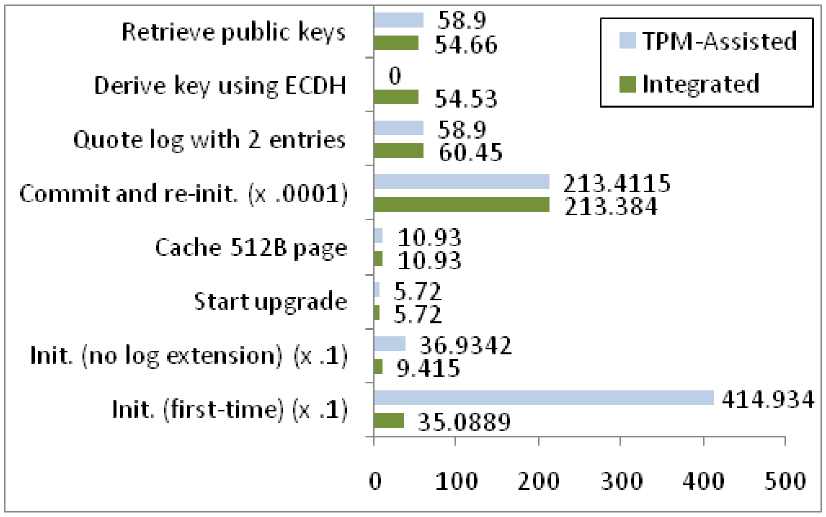
3.3.3 Kernel Firmware

The kernel is invoked whenever the processor resets, and by the application firmware when required. The AVR32 `scall` instruction is used to implement a syscall-style interface between the application and kernel. TinyECC provides software implementations of SHA-1 hashing and Elliptic Curve Cryptography (ECC) [48]. Pseudo-random numbers are generated by Mersenne Twister [49]. These libraries are not significantly optimized for AVR32. Note that the algorithms and key lengths used here may not be suitable for production use in systems with extended lifetimes during which the algorithms may be compromised. A commercial implementation would require a true random number generator. Table 3.1 provides a breakdown of the lines of C++ code in each kernel component. These numbers were generated from the raw source code directories, which include debugging and unused code. We exclude the drivers provided by Atmel.

The kernel consumes 81,312 bytes of program memory. We reserved 88KiB of flash memory to store the kernel code, and another 40KiB to store the persistent data manipulated by the kernel. We set aside 12KiB of data RAM for the kernel comprising 10,872 bytes of static data, 392 bytes for the heap, and 1024 bytes for the stack. The memory consumed by the kernel is unavailable to the application, which does impose an added cost if it becomes necessary to upgrade to a larger microcontroller than would have been required without the kernel. In this prototype, the maximum application firmware image size is 191.5KiB. However, commercial kernel implementations will be significantly more compact in both flash and data RAM than our unoptimized prototype, and clever swapping schemes could potentially eliminate the data RAM consumption of the kernel when it is not active. The audit log in this implementation can record up to 107 upgrades and events before overflowing.



a) Elapsed Time (ms)



b) Energy Consumed (mJ)

Figure 3.5: A performance comparison of TPM-assisted and standalone (integrated) CRAESI.

3.3.4 Performance Results

We now compare the energy and time consumed by our firmware-only prototype (integrated CRAESI) to that consumed by an Atmel AT97SC3203 TPM performing comparable operations (TPM-assisted CRAESI). We used a TPM installed in a PC to perform similar operations to those that would be required by TPM-assisted CRAESI if it were actually implemented. The TPM has a supply voltage of 3.3V and relies on an LPC bus connection. We used Digital Multi-Meters (DMMs) that have limited sampling rates (100-300 ms between samples) to measure the energy consumption of both systems. This introduces some error into our calculations, so we have presented an upper-bound on the energy consumed by integrated CRAESI and a lower-bound on the energy consumed by TPM-assisted CRAESI. The time and energy consumed for a variety of operations is presented in Figure 3.5. From the figure, it is clear that the performance of CRAESI is comparable to a TPM executing similar operations, with the exception of the initialization routines that are much more expensive on a TPM for unknown reasons.

The TPM uses a 2048-bit RSA key to sign the PCRs, which provides security roughly equivalent to that of a 224-bit ECC key, superior to the security of the 192-bit ECC keys used in integrated CRAESI [70]. Due to the use of hardware, the TPM RSA signature generation mechanism is roughly as energy consumptive as the ECC software implementation in the integrated design. The Elliptic-Curve Diffie-Hellman (ECDH) key exchange supported by integrated CRAESI would not be supported by TPM-assisted CRAESI, although it could potentially be replaced with equivalent functionality.

The most significant efficiency drawback of the TPM is that it demands 10.6mW when sitting idle. It may be possible to place the TPM into a deep sleep state to reduce this constant burden, but that is not done in practice in our test system, and may have unexamined security consequences. Let us consider the practical implications of this overhead if attestation is performed once per day per meter in an installation containing five million meters. If AT97SC3203 TPMs were installed in all of those meters, they would consume at least 466,908 kWh per year. In contrast, if integrated CRAESI were used instead, it would consume less than 31 kWh per year.

3.4 CAK Correctness and Fault-Tolerance Analysis

We used the Maude model checker to verify that CRAESI actually satisfies critical aspects of the security requirements put forth at the beginning of §3.2 [27]. First, we modeled CRAESI in rewriting logic, which represents transitions between states using rewrite rules. Then, we expressed aspects of the requirements for the design as theorems, which we converted into LTL formulas that were checked using Maude. We discuss the outcome of this process in this section. We did not discover any errors in the aspects of our implementation that we modeled, which increased our confidence that those aspects of the implementation are correct.

The model comprises several objects within modules that roughly correspond to the modules of functionality in the implementation. We verified the correspondence between our C++ implementation and the rewriting logic model by careful manual inspection. This was feasible because of the small size of the implementation code. Originally, we attempted to unify the basis for a model and executable implementation code into a single code base by implementing the design in a small subset of C# and then compiling the Common Intermediate Language (CIL) corresponding to that code into both assembly language and a model [36]. We abandoned this approach when it became clear that a substantial development effort would be necessary to generate sufficiently efficient assembly code. We provide more details on our efforts to use C# in Section 3.5.

When the model is being used to check high-level properties, such as the correctness of the application firmware upgrade operations, it assumes that any operation runs until completion without interruption. However, this assumption does not necessarily hold in the real world, since power supply interruptions can occur and cause the processor to reset in the middle of any operation. We define rewrite rules that model power supply interruptions that can occur at arbitrary times in separate modules and then prove that the system is fault-tolerant in the presence of power supply interruptions in representative scenarios. The power supply interruptions that we model can be caused by the total loss of power to the processor or a voltage reduction that activates a brown-out detector. Note that we assume the brown-out detector is configured to reset the processor when the appropriate voltage threshold is crossed, above which the processor can operate correctly. We assume that such a power supply interruption results in unpredictable data being written to only the page of flash memory, if any, that is being written when the interruption occurs. We contacted Atmel support to validate that assumption.

Name	Description
aborted(ψ)	A static flash filesystem operation was aborted at stage $\psi \in \{PreCritical, PostCritical\}$.
cur-logent-matches-appfw	The current audit log entry corresponds to the firmware image in the application's installed region.
installed($\mathcal{F}$)	The application's installed region is occupied by $\mathcal{F}$.
cached($\mathcal{F}$)	The upgrade region is occupied by $\mathcal{F}$.
halted	The processor is permanently halted.
appfw-active	The processor's program counter points to a location in the installed region.
rollback	The kernel is about to swap the firmware in the installed region and upgrade region.
sff-as-expected	The static flash filesystem is in the expected state assuming that a particular transaction completed in a filesystem with a particular initial state.
sff-inited	The static flash file system finished initializing.
sff-unchanged	The static flash filesystem is unchanged from its initial state.
upgrade-inited($\mathcal{F}$)	The application has cached $\mathcal{F}$ in the upgrade region and has requested that it be copied into the installed region.
upgrade-started	The application is about to begin caching a firmware upgrade in the upgrade region.

Table 3.2: Propositions used in LTL formulas to model check integrated CRAESI design.

Other types of faults may have similar effects to those of the faults just described, and would therefore be handled by the fault-tolerance mechanisms in CRAESI. For example, a soft error or program bug that corrupts a flash page that CRAESI is modifying and then resets the processor before modifying any other data in flash memory would cause damage indistinguishable from that of a power supply interruption from the perspective of our analysis.

A wide variety of theorems could be important, but we have selected the ones that deal with the parts of CRAESI that have the most complex interactions, since these best illustrate the verification methodology and are the most likely places to find errors. The propositions used to check integrated CRAESI are described in Table 3.2.

The first theorem is concerned with the correctness and auditability of application firmware upgrade procedures. To express it, we stipulate that the system

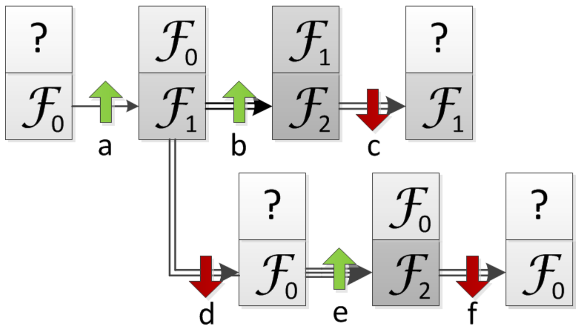


Figure 3.6: Representative, legitimate firmware transitions.

can occupy three primary states when the application is executing, as illustrated in Figure 3.6. The *deployed* state is occupied until an upgrade occurs. The *upgraded* state is occupied after an upgrade has occurred until a rollback occurs. The *rolled-back* state is occupied after a rollback has occurred until an upgrade occurs. To fully explore each of these states, we model transitions between three firmware images that can be installed in order: $\mathcal{F}_0$, $\mathcal{F}_1$, $\mathcal{F}_2$. The content of each firmware image is immaterial. Exactly the transitions and states depicted in Figure 3.6 are correctly permitted by the system. The system can halt in any state. Application activity is represented by the horizontal arrows whose line patterns encode the system state at that point in time. Single lines represent the deployed state, double lines the upgraded state, and triple lines the rolled-back state. The stacked boxes indicate the configuration of the firmware regions in the time represented by the arrows leading away from the boxes. The upper box is the upgrade region and the lower the installed region. An upward-pointing arrow indicates that the application has issued an upgrade request, and a downward-pointing arrow indicates that the kernel has initiated a rollback operation. A question mark in the upgrade region indicates that the state of the upgrade region in the associated configuration is either unpredictable or unimportant.

Theorem 1 *At the conclusion of any operation that modifies the active application firmware image, the audit log is updated to accurately reflect the new state. Additionally, the previous active application firmware image is cached if an elective upgrade is performed (not a rollback).*

Proof. We must verify that *1)* firmware is only modified by explicit firmware upgrade and rollback operations, *2)* those operations can be used to cause only the transitions represented in Figure 3.6, *3)* the audit log accurately represents the history of the system whenever the application firmware is active. We prove this by cases, with each case being encoded as a lemma. The initial state for the proof of each lemma corresponds to a system with an initial firmware image $\mathcal{F}_0$ in the installed region, and the kernel state variables set to the values they would have when the system is first deployed.

Lemma 1 states that the deployed state is stable until transition **a** occurs. Lemma 2 states that transition **a** occurs correctly. Lemma 3 states that transitions **b** and **e** operate correctly. Lemma 4 states that transitions **d** and **f** operate correctly. Lemma 5 states that transition **c** operates correctly. Finally, Lemma 6 states that the firmware audit log is properly updated after every operation. □

Each stand-alone theorem and lemma in this section includes two descriptions, the first in natural language and the second as the LTL formula that was machine-checked.

Lemma 1 *$\mathcal{F}_0$ is installed unless an upgrade operation is performed.*

$$\boxed{\text{installed}(\mathcal{F}_0)\ \mathcal{W}\ (\text{upgrade-inited}(\mathcal{F}_1) \lor \text{upgrade-inited}(\mathcal{F}_2))}$$

This ensures that the initial application firmware on the device is not modified until a specific command to do so is received from the application.

Lemma 2 *If an upgrade to $\mathcal{F}_1$ has been initiated, then $\mathcal{F}_1$ is installed and $\mathcal{F}_0$ is cached by the time the application is activated, and the system remains in that state unless some other upgrade or rollback operation is performed.*

$$\boxed{\begin{array}{l}\text{upgrade-inited}(\mathcal{F}_1) \Rightarrow \\ (\neg\text{appfw-active}\ \mathcal{U}\ ((\text{installed}(\mathcal{F}_1) \land \text{cached}(\mathcal{F}_0))\ \mathcal{W}\ ((\text{upgrade-started} \land \bigcirc(\text{installed}(\mathcal{F}_1) \\ \mathcal{W}\ \text{upgrade-inited}(\mathcal{F}_2))) \lor \text{rollback})))\end{array}}$$

This specifies that $\mathcal{F}_0$ is cached when replaced, and $\mathcal{F}_1$ can be successfully installed at the proper time, and remains unmodified until the application firmware is upgraded to $\mathcal{F}_2$, or it fails to send a heartbeat and is thus rolled back to $\mathcal{F}_0$.

Lemma 3 *If an upgrade to $\mathcal{F}_2$ has been initiated, replacing $\mathcal{F}_n$, and no other rollback operation has yet been performed, then $\mathcal{F}_2$ is installed and $\mathcal{F}_n$ is cached by the time the application is activated.*

$$\boxed{\begin{array}{l}(\text{installed}(\mathcal{F}_n) \land \text{upgrade-inited}(\mathcal{F}_2)) \Rightarrow \\ (\neg\text{appfw-active}\ \mathcal{U}\ ((\text{installed}(\mathcal{F}_2) \land \text{cached}(\mathcal{F}_n))\ \mathcal{W}\ \text{rollback}))\end{array}}$$

This is similar to Lemma 2, but handles transitions to $\mathcal{F}_2$ from either $\mathcal{F}_0$ or $\mathcal{F}_1$.

Lemma 4 *If $\mathcal{F}_0$ is cached at the time that a rollback occurs, then $\mathcal{F}_0$ is installed by the time the application is activated after the rollback unless another upgrade operation occurs.*

$$\boxed{(\text{cached}(\mathcal{F}_0) \wedge \text{rollback}) \Rightarrow (\neg\text{appfw-active}\ \mathcal{U}\ (\text{installed}(\mathcal{F}_0)\ \mathcal{W}\ \text{upgrade-inited}(\mathcal{F}_2)))}$$

This specifies that the application firmware rollback action always operates as expected when rolling back to $\mathcal{F}_0$.

Lemma 5 *If $\mathcal{F}_1$ is cached at the time that a rollback occurs, then $\mathcal{F}_1$ is installed by the time the application is activated after the rollback, and remains installed henceforth.*

$$\boxed{(\text{cached}(\mathcal{F}_1) \wedge \text{rollback}) \Rightarrow (\neg\text{appfw-active}\ \mathcal{U}\ (\Box\text{installed}(\mathcal{F}_1)))}$$

This is similar to Lemma 4, but handles rollback operations that restore $\mathcal{F}_1$. If a rollback restores $\mathcal{F}_1$, then it must be rolling back from an upgrade to $\mathcal{F}_2$, which means that no further upgrades are possible within our model. Thus, this lemma does not include an allowance for further upgrade operations, as is the case in Lemma 4.

Lemma 6 *The current audit log entry corresponds to the installed application firmware whenever the application is active.*

$$\boxed{\text{appfw-active} \Rightarrow \text{cur-logent-matches-appfw}}$$

This states that the latest entry in the audit log is accurate whenever the application is running, ensuring that no undetected actions can be performed by the application. It does not verify the mechanism that is responsible for actually inserting new entries into the log and archiving old entries when the log overflows. That mechanism is a short, isolated segment of code in the implementation that can be manually verified. The primary value of the model checker is in verifying portions of the implementation that interact in complex ways with other portions of the implementation and the environment.

The following theorem is used to verify that the fault-tolerant application firmware upgrade mechanism operates as expected. We modeled non-deterministic power supply interruptions that may occur at any point in the upgrade process. The model checker exhaustively searched all combinations of power supply interruptions, and verified that the application firmware upgrade process always eventually succeeds as long as the power supply interruptions do not continually occur forever. Only one upgrade operation is modeled, because all upgrade operations are handled similarly regardless of firmware content. We tested this theorem on real hardware by pressing the reset button repeatedly during an upgrade and verifying that it still eventually succeeded, but of course we were not able to exhaustively test all possible points of interruption as the model checker did.

Theorem 2 *Executing any application firmware upgrade operation eventually results in the expected application firmware images being cached and installed when the application is subsequently activated, regardless of how many times the processor is reset during the upgrade process, if the processor does not continually reset forever.*

$$\boxed{\neg\Box\Diamond\text{rebooted} \rightarrow (\neg\text{appfw-active}\ \mathcal{U}\ (\text{installed}(\mathcal{F}_1) \wedge \text{cached}(\mathcal{F}_0)))}$$

The initial state for the model checking run of Theorem 2 represents the system running application firmware $\mathcal{F}_0$ after an upgrade to $\mathcal{F}_1$ has been cached and is about to be committed.

The following theorem is used to verify that the fault-tolerant persistent configuration data storage mechanism used by the kernel exhibits correct behavior. As in the previous theorem, non-deterministic power supply interruptions are modeled at every transition point in the model. We model only a single store-commit sequence, because all persistent data is handled identically regardless of identity and content. We tested this theorem on real hardware by setting breakpoints at critical locations in the filesystem code and forcing the processor to reset at those locations. Again, the model checker provides exhaustive testing, which is superior to our manual tests.

Theorem 3 *The filesystem correctly handles any transaction, regardless of how many times the processor is reset during a transaction, as long as the processor does not continually reset forever.*

Proof. We must show that transactional semantics are provided whether or not the transaction is interrupted prior to a critical point. The critical point occurs when the processor executes the instruction that invalidates γ_2, as shown in Listing 2. Lemma 7 checks transactions that are interrupted prior to the critical point and Lemma 8 checks all other transactions. □

Lemma 7 *Executing on FS any filesystem transaction that is intended to update files according to $\varpi \subset FN \times FD$ results in FS by the time the filesystem is subse-*

quently initialized if the transaction is interrupted prior to the critical point.

$$\boxed{(\neg\Box\Diamond\text{rebooted} \land \Diamond\text{aborted(PreCritical)}) \rightarrow (\Diamond\text{sff-inited} \land (\text{sff-inited} \Rightarrow \Box\text{sff-unchanged}))}$$

Lemma 8 *Executing on FS any filesystem transaction that is intended to update files according to $\varpi \subset FN \times FD$ results in $\varpi \cup \{(\nu, \delta) | (\nu, \delta) \in FS \land (\neg\exists\eta.(\nu, \eta) \in \varpi)\}$ following the successful completion of the transaction if it is first interrupted after the critical point or is not interrupted at all. It must achieve this by the time the filesystem is subsequently initialized or the processor is halted, whichever comes first. The processor must eventually halt.*

$$\boxed{\begin{array}{l}(\neg\Box\Diamond\text{rebooted} \land \neg\Diamond\text{aborted(PreCritical)}) \rightarrow \\ (\Diamond\text{halted} \land ((\text{halted} \lor \text{sff-inited}) \Rightarrow \Box\text{sff-as-expected}))\end{array}}$$

3.5 Alternate Implementation

We explored an alternate approach to implementing CRAESI that was motivated by our verification objectives. We attempted to unify the basis for a formal model and executable implementation code into a single codebase by implementing the design in a small subset of C#, with the intent of then compiling the CIL corresponding to that code into both assembly language and a model. We first discuss general principles and considerations that influenced our decision to make that attempt in this section. We then describe the approach and results of that attempt.

It is important to verify the correspondence between a model and the actual system being modeled, which can be accomplished in various ways. Figure 3.7 depicts the workflow that we actually used to process the source code of CRAESI and to manually create the corresponding model. We used manual inspection to verify that the Maude model corresponds to the C++ source code, but a more rigorous approach would have been to formalize the semantics of one of the implementation languages and then directly prove that the implementation corresponds to the model. Note that a system may actually use several implementation languages even if its developers only write code in a single high-level language. For example, a C++ compiler may generate assembly language which is then assembled into binary machine code. The compiler may also use distinct intermediate program representations internally. A drawback of proving correspondence between implementation code and a manually-created model is that the proof must be manually constructed for each specific program.

Program representations at different levels may vary in how amenable their

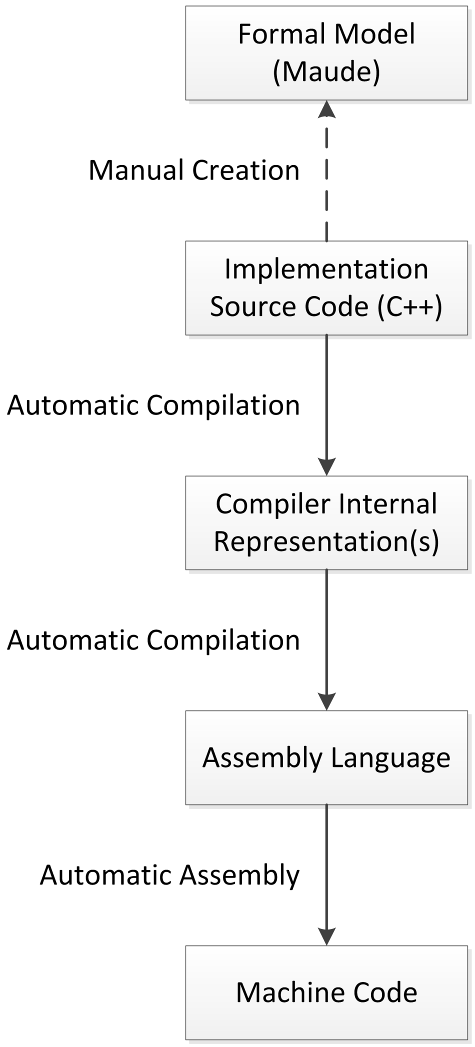


Figure 3.7: Manual formal model generation workflow for CRAESI.

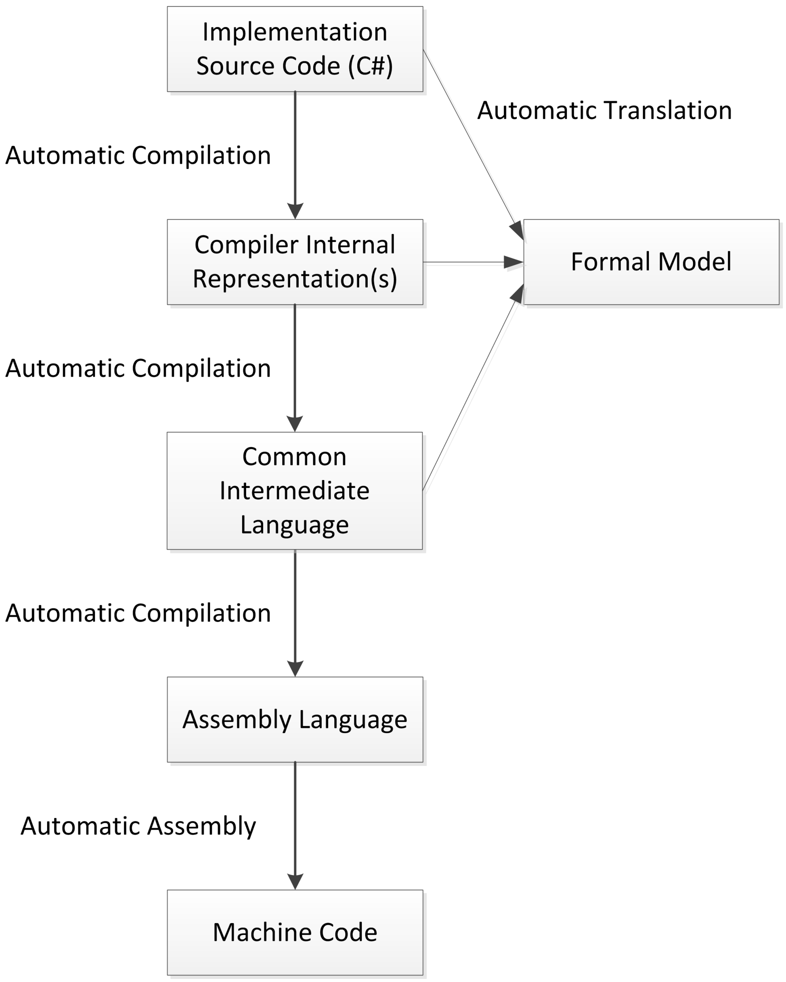


Figure 3.8: Examples of automatic formal model generation workflows.

corresponding models are to formal verification of high-level properties. For example, a high-level language program may contain a loop construct that explicitly processes all elements in a collection, whereas it may be necessary to separately prove that the low-level implementation of that construct actually processes all elements. As another example, the compiler stage that allocates variables to processor registers often reuses registers to hold distinct program variables, making it difficult to track the current values and locations of such variables. On the other hand, the compiler may be able to simplify a complex construct in the high-level language program so that the low-level implementation is easier to analyze.

To eliminate the work involved in manually constructing a correspondence proof for each implementation and model of a program, it may be desirable to automatically translate one of the implementation languages into a model of the program, as depicted in Figure 3.8. However, this assumes that the translator itself operates correctly for the program in question. This assumption must be validated using formal verification to provide a similar level of rigor to that provided by a correspondence proof that was generated manually for a specific program. To provide the practical benefits described above, such a proof of correctness for the translator probably must apply with respect to a large number or all of the programs that can be provided as valid inputs to the translator.

Regardless of the approach used to construct and verify the model, the validity of that model is contingent on the correctness of the toolchain that operates on the code corresponding to the model. An incorrect toolchain may generate lower-level code that does not correspond to the input code. Further assumptions are necessary regarding the correctness of the hardware, etc. Higher levels of assurance can be achieved by validating such assumptions or by shortening the trusted toolchain. Proving correspondence between the model and lower-level code shortens the trusted toolchain by eliminating from consideration the portion of the toolchain that generated that code.

Two observations led us to believe that CIL would be a good representation to use as input to a translator that generates a model of CRAESI. First, CIL is supported by the extensible, open-source Mono compiler for C# [52]. Second, typical CIL code seems to use simpler, more uniform language constructs than code written in a high-level language such as C++ or C# but is still more abstract than AVR32 assembly language. The simplicity of the constructs should reduce the amount of work involved in creating a compiler to generate parts of the model from those constructs. The high level of abstraction compared to assembly language may simplify proofs as explained above. However, exploring the differences between a model generated from CIL and one generated from C# is an interesting area for future research. The challenges that each model would introduce into the process of formal verification are not clear at the outset.

By default, Mono used an interpreter or a Just-In-Time (JIT) compiler to translate CIL into machine code. However, it would not have been feasible to run the entire Mono runtime on an AVR32 processor due to memory constraints, so we used an ahead-of-time compiler instead. The SharpOS project had developed a modular ahead-of-time compiler based on Mono that targeted X86 assembly code [65]. We modified that compiler to target AVR32 assembly code. SharpOS did not fully implement C#, even omitting the garbage collector. Despite this, the accompanying libraries, kernel, and applications included in SharpOS were too large for the AVR32 processor we targeted, so we removed many of their components. We also created preliminary implementations within SharpOS of much of the functionality that was ultimately included in the C++ version of CRAESI. Unfortunately, although we were able to generate valid AVR32 assembly code for the system, the resultant binary image was much too large to fit on the processor. Given this obstacle, we did not attempt to generate a model corresponding to the implementation.

3.6 CAC Design and Evaluation

Flash MCUs with too little memory to fit a full CAK can use a simpler kernel that offloads much of the security functionality to a CAC. The CAC includes cryptographic primitives, limited storage for an audit log of the firmware revisions installed on the main microcontroller, and a communications subsystem for interacting with the kernel. Note that a CAC-based system does not provide security and functionality identical to that of a CAK-based system, but the CAC shares many parts of its design with the CAK. Rather than repeating information from previous sections, we simply discuss where the CAC-based system differs from one based on a CAK.

3.6.1 Security Coprocessor

Offloading security functionality onto a separate security coprocessor introduces additional challenges that must be overcome by the design. First, although our threat model does not address physical attacks in general, the communication channel between the CAC and kernel may be particularly vulnerable to eavesdropping and manipulation by attackers, e.g., using logic analyzers. Furthermore, on the types of microcontrollers we target, it is not possible to prevent the untrusted application from communicating in arbitrary ways with the CAC since the application has full access to the control registers associated with the serial interface. Thus, certain portions of the data communicated between the kernel and CAC are encrypted using a symmetric key that is established when the system is first commissioned. We use 128-bit AES encryption in Counter CBC-MAC (CCM) mode, which provides confidentiality and authentication at the expense of two blocks (32 bytes) of overhead per message [26].

Second, the CAC and main microcontroller can potentially operate in parallel, and a defective kernel may issue commands in an invalid manner when they are not expected by the CAC. Both the timing and ordering of commands may be significant. To address timing vulnerabilities, commands that modify the internal state of the CAC are declared to be non-preemptible. The CAC must not receive any command while executing a non-preemptible command. If this assumption is violated, it indicates a severe error in the system's trusted computing base, and is recorded as such in the audit log. Such an error would either indicate a transient electrical error or the presence of a design or implementation flaw in hardware or software. The latter could require invasive system repair or replacement. Preemptible commands are assigned a lower priority than non-preemptible commands, to ensure that attackers are unable to launch a time-consuming preemptible command that could then block a critical non-preemptible command issued a short time later, possibly preventing a firmware upgrade or compromise event from being recorded. However, a preemptible command is unable to preempt another preemptible command that is already executing. A command is permanently cancelled when it is preempted.

To prevent incorrect command interleavings, the explicit state variable σ_{CAC} is used to determine what commands can be accepted without error by the CAC at each point in time. σ_{CAC} is analogous to σ, but the "Testing Upgrade" and "Waiting for Heartbeat" states are not applicable.

When the CAC is initialized after a reset, it must ensure that any aborted transactions are cleaned up. It does this by checking σ_{CAC}, and if it is in the "Upgrading" state, indicating that the firmware hash accumulator was partially initialized but never committed, it records an "Upgrade Aborted" event in the audit log, to indicate that the main microcontroller's firmware is in an unpredictable state. It then clears the accumulator and transitions to the "Idle" state.

Many of the CAC commands manipulate the audit log or other variables stored in flash. Thus, their operation could be undermined if the CAC lost power after the command was received but before the associated modifications to memory could be completed. Fault-tolerance techniques like those used in CRAESI could be applied in this situation.

The CAC has a simple interface to the main microcontroller that allows the main microcontroller to request attestation operations and submit firmware updates for auditing, as previously described. The firmware provides software implementations of the necessary cryptographic routines, specifically SHA-1 hashing, ECC public-key cryptography, ECDH, and AES-CCM.

The total firmware image running on the CAC requires 24,346 bytes of flash program memory and 820 bytes of EEPROM.

3.6.2 Main Microcontroller

Listing 6 Initialize main microcontroller upon system reset. Network communications between the main microcontroller $\mathcal{M}$ and the CAC C are included.

```
procedure Boot
    $\mathcal{M} \xrightarrow{init} C$
    $\mathcal{M} \xleftarrow{\alpha} C$
    if $\alpha = newkey$ then    ▷ The CAC needs to send a new key. This must only occur once for each system.
        $\mathcal{M} \xleftarrow{\kappa} C$
    end if
    $\upsilon \leftarrow ExternalFlash[UpgWaiting]$
    for $i \in (0, \ldots, |BuiltInProgMem|)$ do    ▷ Handle each page of program memory.
        $\pi \leftarrow BuiltInProgMem[i]$
        if $\upsilon = True$ then    ▷ The application requested an upgrade.
            $\pi_{upg} \leftarrow ExternalFlash[i]$
            if $\pi = \pi_{upg}$ then    ▷ The upgrade firmware page data matches the existing firmware page data.
                $\mathcal{M} \xrightarrow{extend(\{\pi\}_\kappa)} C$    ▷ Extend the CAC's firmware hash accumulator. Encrypt the page data.
            else
                $BuiltInProgMem[i] \leftarrow \pi_{upg}$    ▷ Upgrade the firmware page.
                $\mathcal{M} \xrightarrow{extend(\{\pi_{upg}\}_\kappa)} C$
            end if
        end if
    end for
    $\mathcal{M} \xrightarrow{commit} C$    ▷ Commit the firmware hash to the audit log if it is different from the latest entry.
end procedure
```

The main microcontroller is the coordinator of the entire embedded system. The microcontroller is configured to grant the kernel control over the microcontroller when it is first powered on or subsequently reset. The initialization routine is depicted in Listing 6. First, it attempts to establish communication with the CAC. After acquiring the symmetric key from the CAC if required, the kernel transmits the entire application firmware image to the CAC, which then ensures that the hash of that firmware is the latest entry in the CAC's audit log. The kernel then invokes the application.

The application firmware upgrade process is implemented using an external flash memory, since permitting the application to perform writes to the built-in flash memory (program memory) would permit the application to execute unaudited code and since the data RAM is too small. The application must simply write the new firmware data to the external flash memory and then set a specific location in external flash to a special value. It then resets the microcontroller to invoke the kernel. Every time the kernel starts, it checks that location in flash to see if an upgrade has been requested. Regardless, the kernel then sequentially reads each page of application program memory. If an upgrade has been requested, it then compares that page to the corresponding page read from the external flash. If they differ, the kernel writes the external flash data over the page in the program memory.

3.6.3 Hardware Implementation

Our prototype comprises two interconnected circuit boards, which are depicted in Figure 3.9. The board on the left is the Atmel ATSTK500 prototyping kit with an ATmega644V microcontroller, the CAC. The second board is the Atmel ATSTK600 prototyping kit with a daughtercard containing an ATmega2560 main microcontroller. The ATmega2560 has 256KiB of program memory, but we only use 8KiB for the kernel and 32KiB for the application. Each board has an RS-232 serial port, which is used to implement communication between the main processor and the CAC at 115,200 bps. The STK600 also includes an SPI-accessible Atmel AT45DB041B 4Mbit flash memory chip, which serves as the external flash memory.

The AVR architecture has several characteristics that introduce a variety of security challenges for the kernel. We highlight these by outlining several possible attacks, and also present the countermeasures that the kernel uses to detect or prevent all such attacks.

Installing unaudited code The AVR architecture can be configured to only permit code in the kernel to modify the program memory, but it does not permit the kernel to restrict its entrypoints. The application can directly invoke any instruction

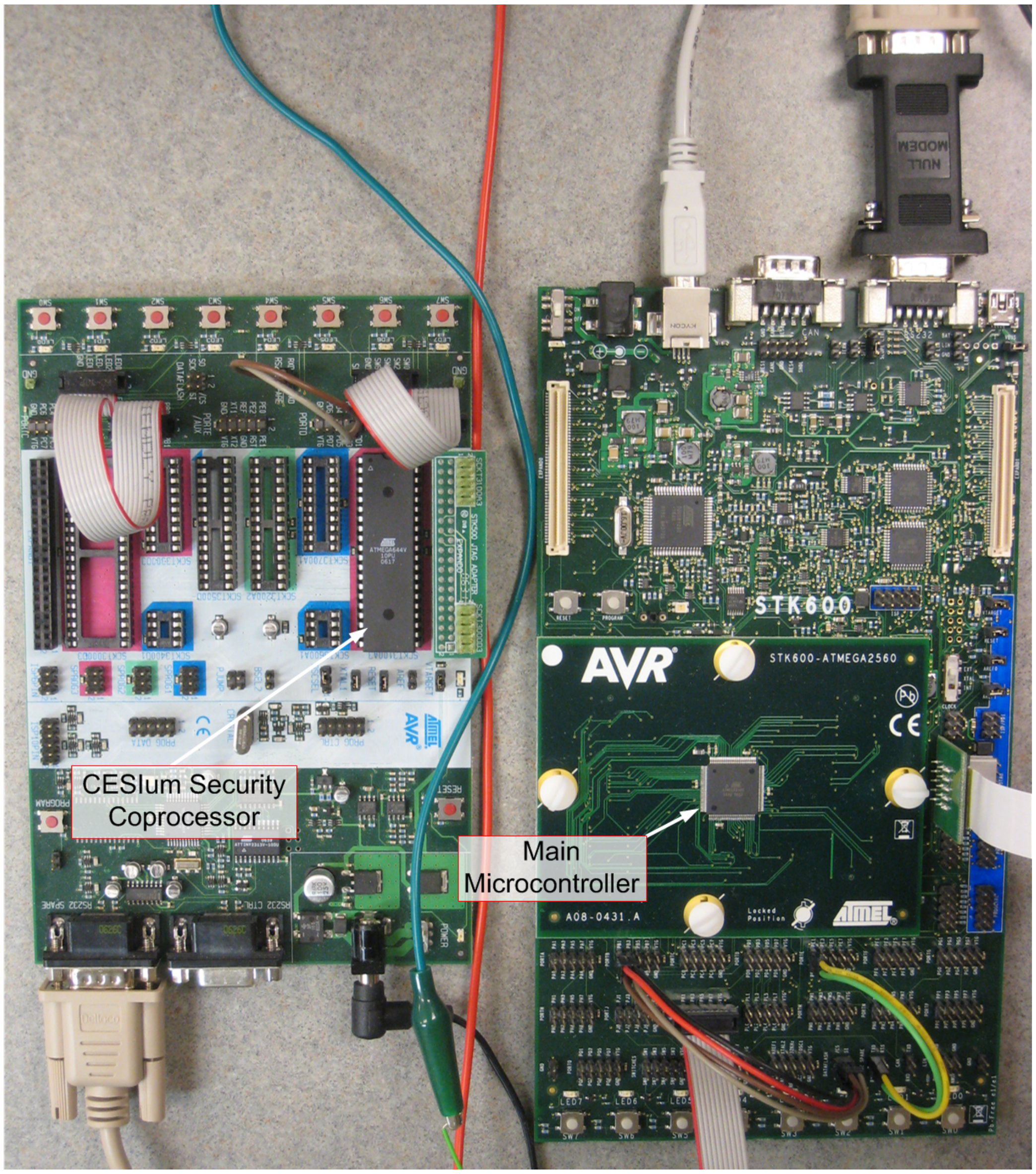


Figure 3.9: Hardware prototype of CESIum.

in the kernel. The kernel necessarily contains at least one `SPM` instruction (for "store program memory") that writes to a location in program memory or performs other flash configuration actions. Thus, a malicious application can directly jump to an `SPM` instruction, bypassing the kernel auditing mechanisms that ordinarily precede firmware upgrades. It may also be possible for the application to manipulate the stack using return-oriented programming so that the application regains control soon after the flash is written [30]. Since the AVR uses both 16- and 32-bit instructions with 16-bit address granularity, it is even possible that attackers could target "gadgets" starting with the latter half of a 32-bit instruction that happens to have the bit pattern of an `SPM` instruction. Similar attacks have been devised for the x86 architecture [64]. Persistent data in the program memory can also be executed as instructions. Such attacks could lead to the execution of unaudited code.

To detect this sort of attack, we actually detect the unauthorized control flow. We devised a lightweight Control-Flow Integrity (CFI) enforcer that is capable of detecting attacks with high probability. More general CFI enforcers have previously been developed for other architectures [2]. The kernel only has a single authorized entrypoint, its reset vector, so all valid control flows must proceed from there. The kernel reset vector copies a 64-bit *CFI secret* to a specific location in data RAM, and the kernel clears all RAM and registers before transferring control to the application to prevent direct disclosure of kernel secrets. Thus, to determine whether a particular control flow is valid, it is necessary to check for the presence of the CFI secret in data RAM. Such a check is also sufficient to validate the control flow iff the CFI secret is actually safe from disclosure.

The kernel must perform a CFI check after any operation that could either lead to the disclosure of a kernel secret or to a modification of the program memory. It is necessary to perform a check after every `SPM` instruction in the kernel, and there are many other security-critical locations in the kernel that we discuss below.

Interrupts could also be used to steal control back to the application. However, this is prevented by setting the AVR lock bits, which are enforced by the hardware, to automatically disable interrupts when control is transferred to the kernel section. These bits are configured as soon as the main microcontroller establishes a key with the CAC, which occurs before the application ever obtains control of the microcontroller. These bits are also configured to prevent the bootloader from modifying its own program memory, which would otherwise permit an attacker to modify the bootloader.

Reading kernel secrets A malicious application could leverage two of the kernel's secrets to perform further attacks. First, it could forge arbitrary firmware measurements if it obtained the symmetric key used to secure communications with the CAC. Second, it could undermine the CFI checks if it obtained the CFI secret. The AVR lock bits also prevent the application from reading kernel program memory, so the only possible way for a malicious application to read the secrets is to leverage instructions present in the kernel itself, using control-flow attacks. To detect such attacks and prevent them from succeeding, the kernel performs a CFI check after each instance of the `ELPM` instruction, which stands for "extended load program memory." That is the only instruction that can directly ready from the kernel program memory. However, other instructions can potentially leave traces of secret data in registers or data RAM, so it is necessary to guard such instructions on a case-by-case basis for each particular kernel. There is another related threat that is more subtle. It is possible for the kernel to contain an `ELPM` instruction in a loop that can be manipulated in such a way as to load the CFI secret into the appropriate region of data RAM and thus pass CFI checks and potentially disclose the entire CFI secret. We place the CFI check inside the loop to detect such attacks. This results in a worst-case scenario of the attacker causing the kernel to load at most one byte of the CFI secret from kernel program memory prior to the CFI check being performed, which does not substantially increase the attacker's probability of passing the CFI check and does not result in the disclosure of any part of the CFI secret.

The attacker could potentially manipulate the very routine that checks the CFI secret in memory in an attempt to discover the secret. For example, consider a routine that loads a copy of the correct CFI secret into memory or registers and neglects to clear that state prior to returning. If the routine were implemented as a loop, the attacker could potentially manipulate the loop control variables to cause the routine to terminate early, before checking the entire CFI secret. Then, the attacker could retrieve the copy of the CFI secret from registers or memory. To avoid such complexities, our prototype does not use a loop, but instead embeds each byte of the CFI secret into a separate comparison instruction. It is still possible that the attacker can gain partial knowledge of the CFI secret with a much higher probability than it would have of guessing the entire secret value at once, by jumping straight to a comparison instruction near the end of the sequence of comparisons. The attacker could then leverage knowledge gained in that way to partially initialize the CFI secret in data RAM prior to performing further guesses.

However, the overall probability of guessing the entire secret using such a process is the same as performing a single guess, since every byte is generated randomly. The attacker must possess the entire CFI secret (except perhaps one byte, as discussed earlier) to avoid detection during control-flow attacks that read other kernel secrets or modify flash memory. Furthermore, any failed guesses will cause the kernel to record an event.

Corrupting kernel state The attacker could attempt to undetectably corrupt kernel state by preventing the kernel from fully initializing, or by manipulating the stack pointer to cause the kernel to overwrite its own data. In the first case, the attacker could initialize registers and memory such that if it subsequently jumped to a location near the beginning of the kernel initialization routine, that routine would initialize the in-memory CFI secret and perhaps selected other kernel memory, but not the entire kernel memory. The attacker could also initialize the locations that are not overwritten to obtain advantage. The kernel leverages the fact that once it has control in its initialization routine, it can maintain that control long enough to verify that the entire memory has been properly initialized after the initialization routine completes.

The malicious application could carefully set the stack pointer prior to transferring control to the kernel so that the kernel overwrites its own data in subsequent operations and potentially compromises its CFI. The kernel places code inline with the security-critical routines to transition to a stack at a fixed location in memory to prevent such attacks from succeeding.

There is a chance that the symmetric key used to secure communications with the CAC contains the sequence of bytes for some arbitrary instructions, including security-critical ones, but the presence and location of such a sequence would be unpredictable to the attacker. It is also straightforward to detect problematic keys at the time they are established if this is a concern.

When an attack has been detected, the kernel sets a flag in EEPROM and immediately forces the processor to reset, since it has few guarantees about how the processor is configured at that point in time. After resetting into a good configuration, the kernel detects the flag in EEPROM and notifies the CAC, which records a special value in the audit log to indicate the event. It then also records the hash value of the modified application firmware. The application also has access to the EEPROM, so it could trigger a false attack notification. However, it never obtains control of the processor between the time that the kernel sets the attack

Module	Lines of Code
Kernel	598
Coprocessor	1487
Crypto	6141
TOTAL	8226

Table 3.3: Lines of C code in each component.

flag in EEPROM and when it notifies the CAC, so it is unable to prevent an attack notification from being sent in that way.

The kernel requires 8,090 bytes of program flash memory out of 8,192 bytes available, and one byte of EEPROM. The kernel only uses data RAM while it is active, so it does not restrict the application's data RAM usage in any way.

We developed a prototype application to demonstrate the upgrade process and to demonstrate the feasibility of performing an attack by jumping straight to a flash programming instruction in the kernel. The attack is detected properly by the kernel.

The firmware for the CAC and the main microcontroller are implemented in a modular fashion, and actually share some code. Table 3.3 provides a breakdown of the lines of C code in each module. These numbers were generated from the raw source code directories, which include debugging and unused code. We exclude the drivers provided by Atmel.

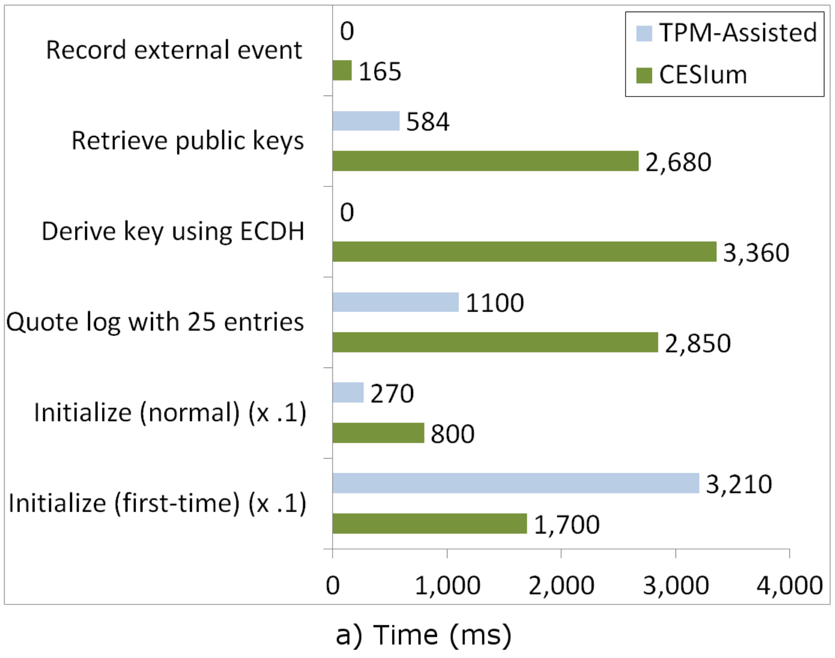
3.6.4 Performance Evaluation

We used a similar experimental configuration to the one we used for evaluating the performance of CRAESI to evaluate the performance of the CESIum coprocessor. The processor was powered using a 2.0V unregulated power supply for these experiments, to reduce energy consumption. Since CESIum is a coprocessor-based system like the TPM-assisted system, it also consumes power while idle. However, it only demands 1.6mW when idle. The experimental results for various operations are presented in Figure 3.10. As for CRAESI, some of operations are not directly comparable. In particular, no direct analogue for the "Record external event" operation in CESIum exists for the TPM-assisted system. These results actually correspond to a slightly different version of the CESIum coprocessor. It was also compiled using an older compiler with different settings, producing a larger binary image. Thus, our performance results are likely pessimistic. We actually exercised the functionality of the CAC by connecting it to a PC over its serial link. The CAC generated its 8MHz system clock using an energy-efficient internal oscillator.

We also tested the timing performance of the main microcontroller using a DMM with a one second sampling interval. We performed a total of five test runs. The process of measuring 32KiB of firmware and performing the corresponding log extension process for the initial firmware took an average of 25.4 seconds with a standard deviation of 0.5 seconds. The average time that the application required to initialize the external flash memory with 256KiB of firmware and the associated command value in preparation for an upgrade was 6.2 seconds with a standard deviation of 0.8 seconds.

3.7 Summary

We presented requirements for cumulative attestation kernels and coprocessors for flash MCUs to audit application firmware integrity. Auditing is accomplished by recording an unbroken sequence of application firmware revisions installed on the system in kernel or coprocessor memory and by providing a signed version of that audit log to the verifier during attestation operations. We have shown that this model of attestation is suitable for the applications in which sensor and control systems are used, and proposed a design for an attestation kernel that can be implemented entirely in firmware. CRAESI is cost-effective and energy-efficient for use on mid-range 32-bit flash MCUs, and can be implemented without special support from microcontroller manufacturers. We used a model checker to verify that CRAESI satisfies important correctness and fault-tolerance properties. The CESIum coprocessor is suitable for use with low-end 8-bit flash MCUs.



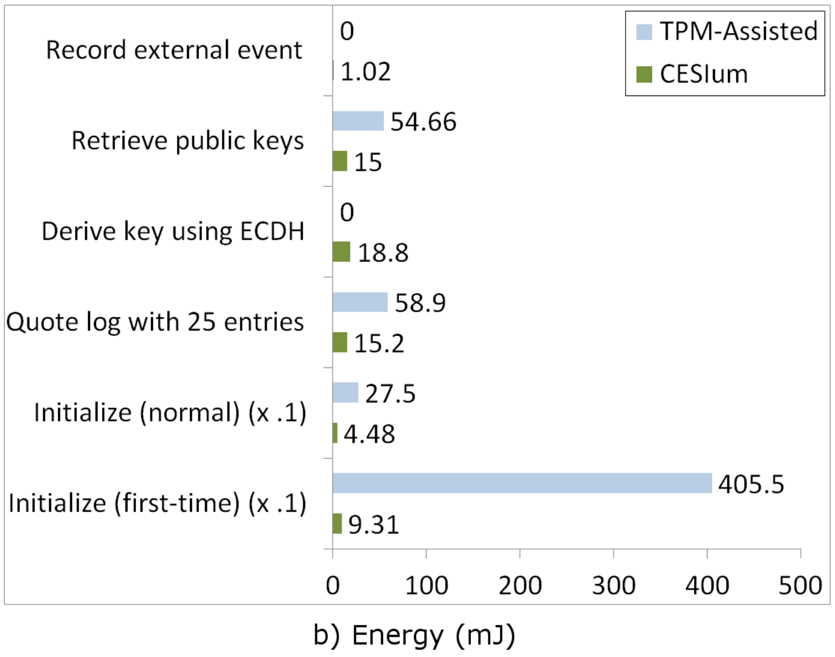


Figure 3.10: A performance comparison of TPM-assisted and CESIum-based remote attestation.

Chapter 4

Architectural Extensions to Support Integrity Kernels

We can improve the isolation, visibility, performance, and compatibility of integrity kernels by enhancing the architecture of host processors to specifically support them. We demonstrate this with a new type of processor called IAP. In this chapter, we first define our threat model in §4.1. It includes powerful attackers, even those that can use Direct Memory Access (DMA) to modify memory without the processor's intervention. Next, we provide a rationale for our design decisions in §4.2, including a detailed argument that IAP detects all attempts to execute unverified memory. We then discuss the implications and limitations of the design, including potential future directions in §4.3. §4.4 presents implementation details for our prototype, which we based on an FPGA with a SPARC soft core. Finally, we summarize the chapter in §4.6.[1]

4.1 Threat Model

We adopt the Dolev-Yao model for attacks on the LAN hosting XIVE [23]. The attacker is permitted to use DMA to modify memory. It is technically possible for the processor to fetch data from peripheral memory regions containing configuration registers that can change their values using mechanisms other than monitored data writes, and interpret those values as instructions. The bus used in IAP specifies when an instruction is being fetched, and we require peripherals to respect those signals and refuse to fulfill instruction fetches from memory regions encompassing anything other than RAM or ROM. We disallow physical attacks in general, but attackers are otherwise permitted to arbitrarily manipulate I/O ports and devices.

[1]This chapter includes material from a previous publication by LeMay and Gunter [44].

4.2 Design

4.2.1 Background on SPARC and LEON3

IAP is based on the LEON3 SPARCv8 soft core by Gaisler Research. We based our design on an instantiation of the LEON3 that implements a 7-stage pipeline, separate data and instruction caches (D-cache and I-cache, resp.), an MMU with a split TLB (D-TLB and I-TLB), and an AMBA 2.0 AHB system bus. Full VHDL source code for the processor is available, so we were free to modify any part of it. Our changes are mostly concentrated in the pipeline, cache, and MMU subsystems.

SPARC is a RISC architecture with a large register file. Registers are organized into a circular set of windows, so that each separate procedure and trap handler is allocated a separate register window that overlaps with the previous one. The overlapping registers can contain parameters and return values. Since only a finite set of register windows are supported (eight, in our configuration), the processor must sometimes swap data between register windows and memory.

SPARC processors support several distinct address spaces, each of which is accessible using 36-bit addresses. Some of the address spaces refer to main memory and peripherals, and others are used to configure the processor or access special features. For example, one address space is used to perform cache flushes. Memory accesses that reference main memory and peripherals and that utilize the MMU are translated to physical addresses according to information contained in the MMU's internal state and page tables. The SPARC Reference MMU (which the LEON3 implements) uses a hardware page table walker, which automatically accesses page tables in main memory until it finally reaches a page table entry that can be used to translate the input virtual address to a physical address. It then stores that translation in a TLB entry so that it can avoid walking the page table while translating any address within the region mapped by the page table entry until that TLB entry is replaced.

Memory accesses by the I-cache are handled specially. Any access by the I-cache initiates a streaming process, whereby additional instructions besides the one requested are automatically loaded from memory. A partial or full cache line is streamed during each operation, depending on a variety of internal conditions.

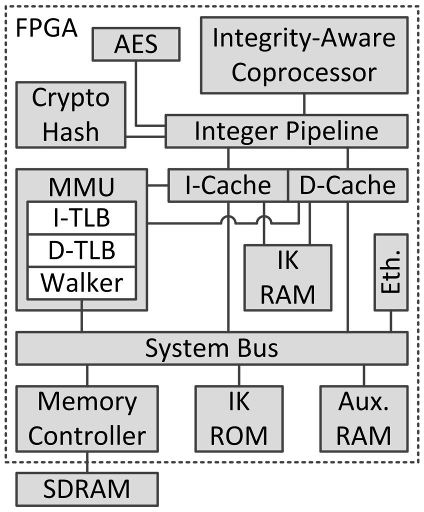


Figure 4.1: Internal connectivity of IAP components.

4.2.2 Integrity-Kernel RAM and ROM

Figure 4.1 illustrates the internal connectivity of the major components in IAP, each of which will be discussed below. The central addition to the processor is a region of on-chip RAM (called the *integrity kernel RAM*) that is only accessible in integrity kernel mode, which is analogous to supervisor mode and possesses strictly greater access privileges. Integrity kernel RAM can be accessed by the I-cache and the D-cache, through a dedicated port for each. Integrity kernel RAM occupies a dedicated address space. Accesses to integrity kernel RAM are not mediated by the MMU since the entire integrity kernel is trusted. IAP contains a ROM from which the integrity kernel is loaded into integrity kernel RAM immediately after the processor is reset. Control is then transferred to the first instruction in the integrity kernel.

4.2.3 Event Handling

For XIVE to be implemented efficiently, it requires hardware support for detecting at least three types of events:

1. A new or recently-modified page of memory being accessed for execution by the pipeline.

2. The program counter being set to a specific value.

Listing 7 *Hardware handling of individual events.*

```
procedure HandleEvent
    $\epsilon \leftarrow$ DetectEvent
    $\alpha \leftarrow$ PrescribeResponse($\epsilon$)
    if $\alpha = \langle Trap, \tau, \delta \rangle$ then
        Trap($\tau, \delta$)
    end if
end procedure
function DetectEvent
    if attempting to execute instruction from page with unset V bit then
        return $\langle HitVBit, \psi \rangle$                    ▷ $\psi$ is the page information.
    else if $PC \in Breakpoints$ then    ▷ Integrity kernel can modify breakpoints.
        return $\langle HitBreakpoint, None \rangle$
    else
        return $None$
    end if
end function
function PrescribeResponse($\epsilon$)         ▷ Determine how to respond to the event.
    if $\epsilon = \langle \tau, \delta \rangle$ then
        return $\langle Trap, \tau, \delta \rangle$      ▷ Other types of responses could be supported in
future versions of IAP.
    else
        return $None$
    end if
end function
procedure Trap($\tau$, $\delta$)                         ▷ Trap to the integrity kernel.
    $ShadowPC \leftarrow PC$
    $ShadowNextPC \leftarrow NextPC$
    $PSR.IntegrityKernelMode \leftarrow True$ ▷ Controls access to processor resources
and causes the I-cache to fetch trap handler code from integrity kernel RAM.
    $PSR.TrapsPreviouslyEnabled \leftarrow PSR.TrapsEnabled$        ▷ Used to restore
$PSR.TrapsEnabled$ when exiting integrity kernel mode.
    (Continue invoking trap handler similarly to native trap handler.)
end procedure
```

3. a specific standard SPARC trap being activated (currently only necessary for prototyping).

Additional event detectors can be incorporated into future versions of IAP to support other types of integrity kernels.

The process for detecting and handling each event in hardware has several phases, as illustrated in Listing 7.

1. The *event detector* activates.
2. The *event filter* communicates any required actions to the processor pipeline.
3. The pipeline perform any required actions.

The following paragraphs describe these phases in reverse, since the earlier phases are increasingly specific to particular types of events.

IAP can currently perform one type of action in response to an event: Generate a trap to a handler in the integrity kernel. Integrity kernel traps are treated differently than native SPARC traps. By convention, an empty register window is always available for a trap handler to use. However, since integrity kernel traps can be processed while a native trap is executing, they can't rely on this window being available. Specifically, this means that the window pointer used internally by the SPARC pipeline is not adjusted in relation to integrity kernel traps. In fact, integrity kernels do not use the register window functionality at all, unless they need to access data in target registers. Instead, they manually save and restore register data using fast on-chip RAM.

This does create a challenge, in that the pipeline must not set local registers with the current and next program counter values as it usually does when invoking trap handlers. Instead, it saves those values in a pair of coprocessor registers. The coprocessor registers can be read and modified by the integrity kernel trap handler, and their contents shadow their corresponding normal registers for "long jump" and "return from trap" instructions. By only shadowing the registers for those instructions, we ensure that the registers can otherwise be used normally by the integrity kernel.

We extended the processor state register with two additional status bits. The *integrity kernel bit* is set whenever the processor is executing the integrity kernel and is used to control access to processor resources and to determine when integrity kernel mode is active. The *traps previously enabled bit* is only set when the processor is in integrity kernel mode and traps were enabled when integrity kernel mode was activated. It is automatically used by the pipeline to restore the *traps enabled bit* in the processor state register when exiting integrity kernel mode. Neither the integrity kernel bit nor the traps previously enabled bit is visible nor accessible in non-integrity kernel processor modes.

The event filter communicates with the pipeline using the coprocessor interface specified in the SPARC standard. That interface standardizes program access to 32 coprocessor registers. IAP event filter register contents can be used as patterns during event detection. For example, the event filter reacts to “instruction executed from address” events whenever an appropriate pattern is specified within the appropriate coprocessor register. It is possible that the 32 coprocessor registers natively accessible by SPARC will be insufficient to store all of the information needed to support future integrity kernels. Thus, IAP implements register windows to expand the register space that can actually be accessed. One of the registers specifies a window index, and an operation on any windowed register actually accesses that register in the window specified by the current value of the window index. Our prototype uses two coprocessor register windows.

Attempts to execute unverified instructions are detected within the I-TLB and the I-cache. Each I-TLB entry contains a *V bit* (for “Verified”) that is cleared whenever the entry is inserted into the I-TLB or the memory region that it encompasses is written. However, TLBs are only consulted when the MMU is enabled, so IAP also includes a separate set of *non-MMU V bits* that each map to a portion of the physical address space. V bits allow an integrity kernel to ensure that specific regions of virtual memory (in this context the physical address space constitutes a separate virtual space) have been verified before any of their contents are executed as instructions. The specific type of verification to be performed must be implemented by each integrity kernel.

The hardware mechanisms for automatically maintaining and retrieving the V bits are described in §4.2.6. IAP also provides facilities so that the integrity kernel can selectively and efficiently set or clear any number of V bits controlling a specific physical or virtual address region in constant time.

Instruction pointer breakpoints are implemented using simple value/mask register pairs and can match either the virtual or physical address of the instruction. Native SPARC traps are detected similarly, but are handled somewhat subtly. When the trap matches, the integrity kernel handler is invoked. It is important that the handler be able to determine the type of SPARC trap that triggered it, so IAP preserves the original trap type in a pipeline register that can be read by a standard SPARC instruction.

4.2.4 Ethernet Access

The Ethernet interface in IAP has been modified to support dual MAC addresses and issue a second hardware interrupt request whenever a packet with the second MAC address is received. It places the packet data into the same DMA buffer that it uses for the first MAC address, which permits the integrity kernel to receive packets without forcing the interface into promiscuous mode or switching between MAC addresses. The separate interrupt can be used to efficiently notify the integrity kernel when it receives packets, even when it is not currently active, although our prototype does not currently use this feature. The auxiliary RAM holds DMA buffers that the integrity kernel can use to interact with the Ethernet interface.

4.2.5 Cryptographic Accelerators

IAP also implements two cryptographic algorithms in hardware, since they are heavily used by XIVE and relatively expensive to implement in software. First, we selected the BLAKE hash routine for its open source VHDL implementation, status as a SHA-3 finalist, and good performance [63]. It is integrated with the pipeline itself, and can be used by executing 32- and 64-bit load instructions, the results of which will then be hashed. We exploit the fact that the LEON3 SPARC implementation only uses the lower five bits of address space identifiers to allow the integrity kernel to specify that data should be hashed by setting the upper three bits of the address space identifier to a specific value. The implementation is sufficiently fast that hashing does not stall the pipeline. After a complete message has been hashed, the final hash value can be retrieved by accessing a specific region of memory in a special address space, which causes the hash to be finalized before returning the value. The finalization process does stall the pipeline for a short period of time, but is only invoked once per message.

Second, 128-bit AES is integrated in the pipeline and encrypts data loaded with a different address space identifier modifier. Its key can be set by storing data to specific locations within a configuration address space. IAP implements parts of Counter (CTR) mode, since it lends itself naturally to the serialized data encryption that is implemented in the pipeline, is popular in higher-level cryptographic modes, imposes little additional hardware cost, and can be easily used to implement Electronic Codebook (ECB) mode encryption. To generate ciphertext, CTR mode sequentially XORs words of loaded memory data against words of a counter value processed using AES. To implement ECB mode using CTR mode, the integrity kernel must first manually set the counter value to a block of plaintext. Then, it must load data from a static block of memory containing zeroes, which causes the XOR transformation to simply load the raw AES output into the specified register. The AES implementation can cause pipeline stalls. However, it mostly operates in parallel with the pipeline, so stalls can be avoided by inserting sufficient instructions between the point at which each AES operation is initiated and the point at which its output is used.

4.2.6 Coverage of Pipeline Instruction Inputs

The circuitry that fetches instructions for the pipeline is complex, and it may not be immediately obvious that IAP correctly requires all instructions to be verified before being executed by the pipeline. This section argues that it does so by presenting a detailed analysis of IAP's interactions with the fetch circuitry. Despite the complexity of this portion of IAP, note that IAP effectively removes other components of the processor from the integrity kernel's TCB. Specifically, the specialized integrity kernel isolation mechanism in IAP isolates the integrity kernel by default, whereas MMU-based isolation must be correctly configured to be effective. Furthermore, the mechanisms described in this section associate verification data with structures that are populated by the page table walker in the MMU, which removes the page table walker from the integrity kernel's TCB.

To provide background for the following analysis, we also explain the components within the processor that are capable of issuing system bus write requests. Recall that our threat model only considers instructions fetched from memory regions that are populated by such components. The relevant connectivity of the system is depicted in Figure 4.2. The direction of data flow within the processor core is indicated by arrows. Note that certain peripherals contain both bus master and slave interfaces, such as the Ethernet interface.

The D-cache is responsible for storing data from the pipeline into system bus memory regions. A variety of sequences of events can occur as a result of a write to the D-cache. When the MMU is disabled, the D-cache immediately issues a write request to the core bus interface. The core bus interface arbitrates between all memory clients within the processor to serialize their accesses before issuing them to the system bus controller. The interface uses a static priority scheme to select

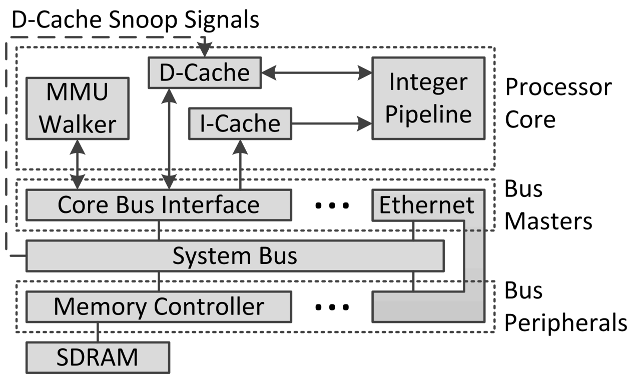


Figure 4.2: Connectivity of potential instruction data sources.

between multiple outstanding requests. I-cache requests are top priority, followed by requests from the D-cache, and finally the MMU page table walker. When the MMU is enabled, the address must be translated before a write request is issued. The MMU page table walker is also capable of modifying memory, because it needs to update bits in page table entries when memory is accessed. Outside the processor core, the Ethernet interface operates as a system bus master, and thus can freely write to memory.

The D-cache already monitors all activity on the system bus, since it uses snooping to update cache entries. IAP extends that monitoring circuitry to also clear each V bit whenever a system bus master (including the processor core) writes to a region encompassed by that bit. If the affected V bit is inside an I-TLB entry, it is cleared in the same cycle that the system bus request is executed on the bus. The updated bit state will be available to TLB requests fulfilled after that cycle. The non-MMU V bits take one additional cycle to update, since their old values must first be read from on-chip RAM, updated, and then stored back to the RAM. Simultaneous read and write requests to the RAM cause the read request to return the old value. The timing of these events is depicted in Figure 4.3a.

The pipeline issues a fetch request to the I-cache by sending it the virtual address of the desired instruction. The I-cache can process the request in one of several ways, depending on its current configuration and internal state. The simplest scenario is when the instruction is already present in the cache. The only way to modify data in the I-cache is to use a diagnostic channel, but IAP disables that channel. Thus, all instructions in the I-cache have been previously verified.

The I-cache transitions directly into streaming mode when a cache miss occurs with the MMU disabled, as shown in Figure 4.3b. Streaming mode attempts to

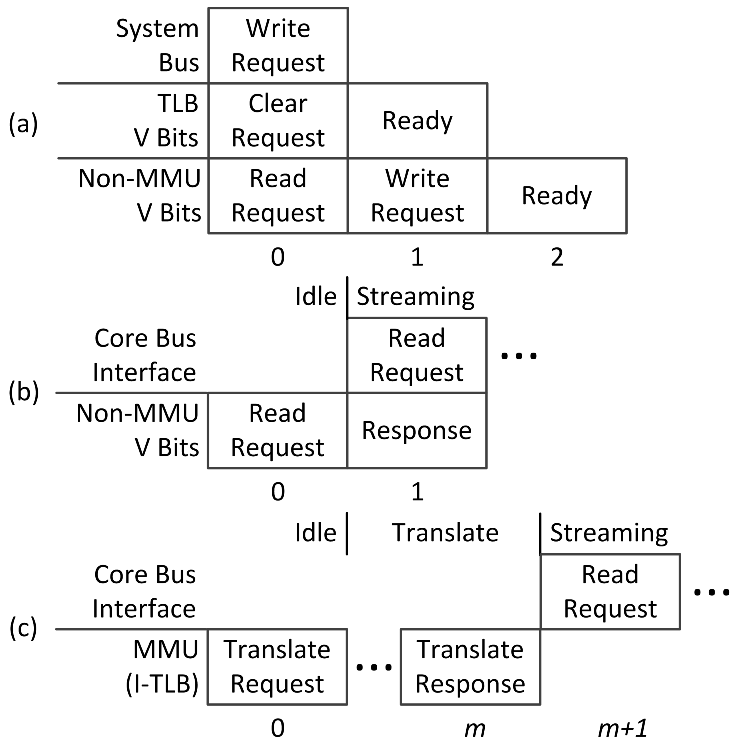


Figure 4.3: V bit update timing.

load a partial or whole cache line of instructions into the I-cache. The appropriate non-MMU V bit is consulted during that transition, and is processed by the IAP event detector. Since the non-MMU V bit only reflects writes that occurred at least 2 cycles prior to the V bit read request, we specially handle the corner cases that fall inside of that bound. The V bit read request is issued in the cycle prior to the first cycle in streaming mode. To detect system bus writes to the region about to be streamed that occur 2 cycles prior to streaming (when cycle 1 of Figure 4.3a coincides with cycle 0 of Figure 4.3b), the I-cache monitors the non-MMU V bit write requests issued by the D-cache. If a V bit write request is issued in that cycle to the location that the I-cache is requesting, the updated V bit from the D-cache is used, and the V bit read request is canceled.

Writes to the desired cache line one cycle prior to streaming through the end of the streaming operation (mid-stream updates) are prohibited and result in a fetch error. Mid-stream updates can in fact be performed without generating errors on the stock LEON3. However, they ought to be discouraged, since some of them have varying results depending on the configuration of the processor's cache parameters.

The I-cache detects mid-stream updates by monitoring the V bit request channel of the D-cache. Attempts to clear the V bit for an address within the I-cache's current cache line trigger an error. It is necessary to check both the current V bit clear requests as well as those from the previous cycle during streaming mode, in case cycle 0 of Figure 4.3a coincides with cycle 0 of Figure 4.3b. If a legitimate program performs a mid-stream update, its desired effects can still be emulated by a trap routine in the target OS. Future versions of IAP may handle mid-stream updates like any other write to an executing page of memory, but that could increase the complexity of the hardware and the potential cost currently seems unwarranted for such unusual software constructs.

When the MMU is enabled, cache misses are first checked against the I-TLB. If an I-TLB entry is hit, its V bit is provided to the IAP event detector and the I-cache transitions to streaming mode. Otherwise, the MMU walks the page table and inserts a new entry in the I-TLB before transitioning to streaming mode. The V bit of the new I-TLB entry is initially unset. The timing of these processes is denoted in Figure 4.3c. The streaming cache line modification detector described above is also active when the MMU is enabled. It is necessary to check both the current V bit clear requests from the D-cache as well as those from the previous cycle during streaming mode, in case cycle 0 of Figure 4.3a coincides with cycle *m* of Figure 4.3c.

4.2.7 XIVE Network Protocol

There are three requirements that the network communications protocol between the integrity kernel and the approver must satisfy:

1. **Security**, to prevent eavesdropping and to detect attempts to modify packets.
2. **Low latency**, to minimize the amount of time that the end node is blocked waiting for a response from the approver.
3. **Simplicity**, since it must be implemented in the integrity kernel, which has a very small codebase.

We constrain the approver to occupy the same local area network as the end node to minimize latency. This permits us to define a link layer protocol, called the *XIVE Network Protocol (XNP)*. Each XNP packet is 96 bytes long (excluding the 14 byte Ethernet header and four byte Ethernet CRC), and can be represented formally as a tuple $\langle \nu, \tau, \varsigma, \phi, \mu \rangle$ (excluding the Ethernet header). To prevent replay attacks, both approver and kernel nonces ν are drawn from a single, strictly increasing sequence. The packet type $\tau \in \{boot, verify, exit\} \times \{request, response\}$. The sequence number ς is used to match responses to requests. The composition of the payload ϕ is specific to each type of exchange described below. EAX mode is used to encrypt and authenticate $\langle \nu, \tau, \varsigma, \phi \rangle$ with AES, and the resultant MAC value μ is then appended [11]. The XIVE kernel implements logic to resend packets after some timeout until a response is received. We now discuss specific XNP exchanges between the XIVE kernel $\mathcal{K}$ and the approver $\mathcal{A}$. Only the payload ϕ is depicted (when non-empty), but each transfer actually involves a complete packet.

The approver resets its internal representation of an end node's state, which we discuss later, whenever it receives a boot request from that node. The XIVE kernel also uses the boot request to locate the approver by sending it to a specific Ethernet multicast address whenever it initializes after a reset. It then stores the MAC address of the approver for use in addressing all future transmissions. Using a multicast group simplifies network configuration. It does not lead to vulnerabilities potentially posed by a malicious host impersonating the approver, because of the protection afforded by EAX using pre-shared keys.

The XIVE kernel issues a page verification request whenever it detects an attempt to execute instructions from a page with an unset V bit: $\mathcal{K} \xrightarrow{\langle \tau_p, \gamma, \beta, \theta \rangle} \mathcal{A}$, where the page type τ_p is derived from the size of the page, the state of the MMU (enabled/disabled), and the processor mode (supervisor/user), γ is a unique identifier for the currently-executing process, β is the virtual base address of the page, and θ is the result of hashing the entire page using BLAKE. The XIVE kernel blocks the execution of the processor until it receives a response from the approver: $\mathcal{K} \xleftarrow{\langle \alpha \rangle} \mathcal{A}$, where α specifies what action the XIVE kernel must take next, which can be either to terminate the process or to resume its execution.

The XIVE kernel issues an exit request when it detects that a process context is about to be destroyed: $\mathcal{K} \xrightarrow{\langle \gamma \rangle} \mathcal{A}$, where γ is the same process identifier used in the page verification requests. This permits the process identifier to subsequently be reused. Note that this does introduce a certain level of trust in the target OS to achieve the full assurances possible with XIVE, since an OS that causes XIVE to issue an exit request without actually destroying the appropriate context can then potentially be permitted by XIVE to execute programs that contain unapproved combinations of pages. However, all code that is executed must still be recognized and approved.

4.2.8 Approver

The approver's roles include generating and maintaining a whitelist, a database of pre-shared AES keys, and a representation of the internal state of each active end node on the network. Our initial prototype is simplified in that it only communicates with a single end node.

To maintain a partial representation of the internal state of each end node, the approver creates an empty state when it receives an XNP boot request packet and updates that state when each new page verification request or process exit request packet is received. The current state of each node n can be represented as a function $\sigma : \Gamma \to 2^{\Pi}$ generated in response to a set of "current" page request packets ρ received from n, where Γ is the set of all context numbers that are contained in ρ and Π is the set of all approved programs:

$$\sigma(\gamma) = \bigcap_{\langle \tau_p, \gamma, \beta, \theta \rangle \in \rho} \left\{ \pi \in \Pi \,\middle|\, \langle \tau_p, \beta, \theta \rangle \in \pi \right\}$$

where $\langle \tau_p, \beta, \theta \rangle \in \pi$ iff the page with the specified characteristics is contained within program π. When $\sigma(\gamma) = P$, it means that the process identified by γ has only previously executed code that is contained in all programs in P. P may contain many programs, since different programs can contain pages of code with identical characteristics. A page request packet $\langle \tau_p, \gamma, \beta, \theta \rangle$ loses currency and is removed from ρ when the approver receives a process exit request packet $\langle \gamma \rangle$. We say that n has entered an unapproved state as soon as $\exists \gamma.\ \sigma(\gamma) = \emptyset$, meaning that the process identified by γ is not a recognized program.

To generate a whitelist, the approver can be operated in learning mode, in which it approves all pages of code and outputs a database representing all programs that were executed by n.

The approver software is implemented as a multi-threaded C++ program. One thread performs administrative functions, another receives, decrypts, and verifies packets, a third processes the received packets with respect to the system state database, and the final thread encrypts, authenticates, and transmits newly-generated packets.

4.2.9 XIVE Kernel

The kernel was coded entirely in assembly language and is described in Listing 8. It has 859 instructions and uses 496 bytes of data storage in integrity kernel RAM. In an auxiliary RAM it also uses 648 bytes for Ethernet DMA buffers and 2304 bytes for the hash cache.

Listing 8 *XIVE kernel.*

```
procedure Boot    ▷ Obtains control immediately after every processor reset.
    $\eta_{dc} \leftarrow$ AddressOf(destroy_context)
    InitBP($\eta_{dc}$)    ▷ Initialize breakpoint to detect process context destruction.
    InitAES($\kappa$)    ▷ Initialize AES using key shared with the approver.
    XNPBoot    ▷ Perform XNP boot exchange.
    $\eta_{to} \leftarrow$ AddressOf(target OS)
    Jump($\eta_{to}$)    ▷ Transfer control to target OS.
end procedure
procedure HandleBreakpoint
    SaveLocalRegisters
    XNPProcessExit($\gamma$)    ▷ Perform XNP process exit exchange.
    RestoreLocalRegisters
end procedure
procedure HandleUnsetVBit($\psi$)    ▷ $\psi$ is the page information.
    SaveLocalRegisters
    SetVBit($\psi$, True)    ▷ Doing this first ensures that any DMA accesses
that occur during the subsequent verification operations are detected when the
processor resumes normal execution.
    $\theta \leftarrow$ BlakeHash($\psi$)    ▷ Hash entire page.
    $\alpha \leftarrow$ XNPVerifyPage($\psi, \theta$)    ▷ Perform XNP page verification exchange.
    if $\alpha$ = resume then
        RestoreLocalRegisters
        (Resume process execution.)
    else
        HaltProcessor▷ Our prototype simply halts the target when it enters an
unapproved state, rather than attempting to selectively terminate the unapproved
program.
    end if
end procedure
```

The kernel initially sets a breakpoint to detect OS-level process exit events. It also contains a handler for attempts to execute unverified memory. Each of these handlers will be described below. After that, it installs the AES key that it shares with the approver. Next, it performs the initialization step of XNP. Finally, it transfers control to the main OS. The rest of this section discusses each of the trap handlers and their supporting functions.

The unverified execution trap handler is executed each time a region of memory is executed for the first time or has been modified since it was last verified. It first sets the appropriate V bit. Doing this first ensures that any DMA accesses that occur during the subsequent verification operations are detected when the processor resumes normal execution. It then hashes the entire page of memory using BLAKE. Next, it sends an XNP page request to the approver. It uses the current MMU context number as the context value that is sent to the approver. After the trap handler receives a valid response from the approver, it resumes the processor's normal execution. To simplify its implementation, our prototype kernel simply halts the target when it enters an unapproved state, rather than attempting to selectively terminate the unapproved program.

Physical pages of kernel code and shared libraries are mapped into multiple virtual address spaces for different processes, so XIVE by default hashes and verifies them in each address space. To reduce this unnecessary overhead, we implemented a hash caching mechanism that stores the hash for each I-TLB entry at the time that it is calculated. Subsequent invocations of the verification routine check each entry in the cache prior to calculating the hash. We extended IAP with the ability to read the V bit and valid bit associated with each I-TLB entry. If both of those bits are set for an I-TLB entry associated with a hash cache entry that was generating from the exact physical memory page currently being verified, the pre-computed hash value is used. The hash cache-checking logic does introduce additional overhead, but also commonly reduces the number of hash operations that are required. Additional optimizations to reduce verification overhead should be explored in the future. Our prototype uses 2304 bytes of auxiliary RAM for the hash cache, although this is in fact vulnerable to manipulation by the target OS. A deployable implementation would place the hash cache in integrity kernel RAM.

The process exit trap is triggered when the pipeline attempts to execute a specific instruction in the kernel function that destroys a process context. It loads the affected context number from the structure passed into the kernel function and then performs the appropriate XNP exchange each time it is triggered.

Whenever a trap handler is entered, it first saves all of the local registers and the processor state register to integrity kernel RAM. This permits the handlers to freely use the local registers as well as instructions modifying the condition codes in the processor state register. This means that the handlers have eight registers to manipulate, which are insufficient for some operations. Thus, additional blocks of memory are reserved to implement a pseudo-stack for register swapping. The kernel does not implement an actual stack, because it is simpler to directly address the reserved memory in the few instances that it is required.

The target-aware Ethernet driver is a significant source of complexity in the integrity kernel. The integrity kernel shares the Ethernet interface with the target OS, but it can't rely on the target having configured it in a specific manner. So, each time the integrity kernel sends a packet, it first checks to see if the target has initialized the Ethernet interface. If so, it redirects the Ethernet interface to a new transmit buffer in auxiliary RAM, but uses the receive buffer configured by the target OS. Otherwise, it also redirects the receive buffer to auxiliary RAM.

By retaining the target-configured receive buffer, the integrity kernel ensures that packets intended for the target that arrive while the integrity kernel is active are still eventually received by the target. The integrity kernel simply processes all packets that are intended for itself and restores their descriptors before resuming the target. However, the integrity kernel does perturb the system in that its own packets can become interleaved with those destined for the target, introducing "holes" in the receive buffer. The Gaisler Ethernet driver in Linux by default does not scan the entire receive buffer when it receives an Ethernet interrupt, so its operation is disrupted by this behavior. However, it was a simple matter to modify the Linux driver to scan the entire buffer upon receiving each Ethernet interrupt.

4.2.10 Target OS and Applications

Linux 2.6.36 serves as the target OS in our prototype, hosting a Buildroot userspace environment. It was necessary to slightly modify the target system to make it compatible with XIVE. Beyond the modifications discussed in §4.2.9, it was necessary to modify the Linux kernel to allocate smaller pages of memory containing kernel code. By default, the entire kernel space was placed into a single 16MiB mapping. Since many parts of the kernel are dynamic, this made it infeasible to generate a comprehensive set of whitelist entries for the kernel and was also necessitating an enormous number of verification operations. So, we break the 16MiB kernel region into a set of contiguous 256KiB pages soon after the kernel begins booting. We also modified the alignment of kernel code and data sections at link-time so that code and data pages are not mixed.

An additional problem with the default Linux kernel is that the LEON3 architectural support routines flushed the entire TLBs and caches very frequently. This was predictably generating large numbers of re-verification requests to the approver that hurt performance. To resolve this problem, we implemented selective flushing by address regions for both TLBs and the I-cache in the hardware and the Linux kernel. Many architectures already include such hardware functionality and kernel support.

Userspace code also presented challenges for XIVE, because programs and libraries mixed code and data pages. We modified the linker scripts to re-align the program sections and thus avoid that issue. A secondary issue was caused by the particular way in which Linux implements dynamic linking of programs with shared libraries. It uses a structure, called the procedure linkage table, which is a region of memory that is gradually filled in with jump instructions to shared library routines. By default, it is gradually populated on-demand. This results in frequent modifications to an executable page of code, which inflates the size of the whitelist and generates additional traffic to the approver. To resolve this issue, we configured all programs at link-time to preemptively populate the entire procedure linkage table when they are first launched.

We optimized the context switching mechanism to cause the processor to switch back to a dedicated kernel context upon entering the kernel. The I-TLB and I-cache include the context number in their entries' tags, so this reduces pressure on those structures.

For the purposes of our experiments, the Linux kernel and the userspace root filesystem are merged into a monolithic memory image that is loaded into main memory over JTAG before the system boots. To permit the construction of a whitelist, we cause the image to fill a whole 16MiB page, as initially used by the kernel, and zero-fill all unused space within the image.

4.3 Discussion

4.3.1 Deployment and Management

In any XIVE-protected environment, the following elements must be deployed:

1. IAPs to operate all programmable portions of the end nodes.

2. At least one co-located approver server that is statically configured to be resistant to attacks, since it is unable to rely on XIVE protection itself.
3. An integrity kernel ROM image for each IAP.
4. Pre-shared keys to permit authenticated, encrypted communication between each end node and the approvers.

A variety of protocols can be devised to install keys and ROM images in end nodes, and may resemble the protocols that have previously been developed to securely configure sensor nodes, such as SCUBA [59]. In that example, software attestation could be used to establish a tiny trusted environment on the end node that then communicates with the pre-installed integrity kernel and uses a default key to authenticate itself to the integrity kernel. It could then install a new key. This process could be repeated whenever the key needs to be updated, using the appropriate key for the initial authentication. If authentication failed during this process, indicating that an attacker may have changed the key used by the integrity kernel, the administrator would be alerted to the fact and could initiate physical remediation procedures. A similar process could also be used to perform and verify integrity kernel ROM upgrades, assuming that a default ROM image is initially installed.

4.3.2 Data Integrity

XIVE does not enforce the integrity of all objects within the target OS. It enforces the integrity of executable code, which can be used as a basis for enforcing the integrity of other system objects, but it does not directly provide such protections. Other systems have been proposed for protecting the integrity of data. For example, Karger discussed the limitations of discretionary access control and proposed a finer-grained discretionary mechanism to address the threat of trojan horse programs abusing the permissions afforded to a specific user to maliciously modify files [38]. It relies on a database containing information about what sorts of data flows specific applications are expected to construct based on their input parameters and uses a trusted path to the authorized user to grant additional permissions on-demand.

Other policies with similar objectives have been devised. Biba proposed a variety of mandatory and discretionary access control policies that prevent low-integrity data, computations, and subjects from influencing high-integrity or incomparable data, computations, and subjects [14]. The mandatory controls proposed therein are based on a lattice model of integrity levels derived from military applications. Clark and Wilson proposed a different access control policy intended for commercial applications [20]. It associates each program with specific data that can be modified by the program when that operation is requested by a specific user. It requires that each program be certified to properly transform all of its allowable input data to valid output data. It also specifies that special programs should be used to transform unconstrained data, such as user inputs, into allowable input data for specific programs. Karger proposed implementing their model using the SCAP secure capability architecture that was envisioned to provide hardware support for capabilities, and discussed the practical challenges associated with such an endeavor [39].

4.3.3 Denial-of-Service Attacks

Since XIVE relies on network communications to approve the forward progress of each end node, attackers can deny service to legitimate users of those nodes by interfering with XNP. However, XNP only operates on the LAN, which reduces the ability of attackers outside of the LAN to launch Denial-of-Service attacks.

4.3.4 Control Flow Attacks

Control flow attacks can succeed without injecting new code into the target system [16]. Thus, XIVE does not prevent them. However, some types of control flow attacks, such as return-oriented-programming, can be prevented using address space layout randomization [13]. Each XIVE whitelist is specific to a particular address space layout. However, XIVE could potentially be adapted to work with randomized address spaces by causing the approver to issue a seed to control the randomization process on an end node. That seed could then be used by the approver to translate page verification requests.

4.3.5 Bytecode Support

Programs written in Java, .NET, and other languages can be distributed as bytecode, which poses a challenge for XIVE. Bytecode is never directly executed, but is instead processed by an interpreter or a JIT compiler, the output of which is ultimately executed. Currently, XIVE would simply verify instructions executed within an interpreter, a JIT, and the output from the JIT. Certainly, it is desirable to verify interpreters and JITs, but it is likely to be infeasible to whitelist JIT outputs, since the JIT may dynamically generate various instruction streams. This can be handled by monitoring data reads and writes by recognized JITs, so that bytecode inputs can be verified and the output instruction streams intended to be executed in the future can be excluded from verification. Patagonix includes some elements of this approach [47]. Similar considerations apply to other programs that are processed by JITs, such as those written in JavaScript.

4.3.6 Multicore

One potential strategy for adapting XIVE to a multicore environment is to replicate most of its functionality on each core, designate one instance as the leader, and create local communication channels that connect all instances. Then, whenever an instance needed to communicate with the approver, it could route the communication through the leader, which would be the sole instance with access to the Ethernet interface. The leader instance must be co-located on the core with the Linux Ethernet driver, to prevent contention.

4.3.7 Alternate Usage Models

It would be a simple matter to adapt the approver to approve rather than deny by default, and thus enforce a blacklist of known malware, permitting all other software to execute.

Alternately, by simply approving all software that is measured by the end node on the approver rather than preventing the execution of non-whitelisted software, the approver could then field remote attestation requests on the behalf of the end node. Some advantages of this approach over conventional remote attestation, such as that implemented by the Linux Integrity Measurement Architecture [58], are that audit logs are maintained in a central location to further reduce the TCB on end nodes, cumulative attestation can easily be provided [43], it conclusively reveals the presence of malware on infected target systems since malware is unable to block attestation requests, and the use of public-key cryptography is centralized so that fewer nodes must be upgraded if it is eventually broken.

4.3.8 Potential Applications for XIVE

Ideally, XIVE is applicable to any system based on IAP. However, it currently relies on whitelists of allowed kernels and programs, which are only feasible to construct for certain environments. It also requires sufficient connectivity between target hosts and approvers to execute XNP. We focus on three particular environments that are popular, security-critical, and amenable to whitelist construction.

Many data centers are well-suited to XIVE. They are centrally-administered, run a slowly-changing set of programs, and have high-speed connectivity between hosts. They exist in facilities with strong physical security, and primarily face threats from remote attackers that can only influence the data centers' network traffic.

A variety of Process Control Systems (PCSs) are also well-suited to XIVE. For example, Intelligent Electronic Devices (IEDs) are used in electric substations to monitor and control the flow of electricity. They contain upgradeable computers, so it is possible that they could be compromised with malware. Their operating environment shares most of its characteristics with that of data centers, but it has different external connectivity. Substations communicate over FANs that are accessible from few facilities. Some substations are also accessible over maintenance links. However, both types of networks can potentially be compromised and used to propagate malware, and can also enable attacks by insiders.

AMI is another important type of PCS that can be protected by XIVE, since meters contain upgradeable firmware. Meters communicate over a wide variety of private FANs, ranging from low-speed wireless links to broadband connections. However, they also contain very simple software, so they require little communication to verify all of it. Meters operate with minimal physical protections, so it may be unrealistic to prevent all physical attacks on them. However, meter compromises are potentially most dangerous to the stability of the power grid when a large number of meters is compromised, so preventing large-scale remote compromises of meters is a valuable objective. Non-physical attack vectors include I/O ports on the meter (such as short-range IR ports), HANs that are intended to communicate with appliances associated with the metered premise, and compromised FANs. HANs may be used to connect a wide variety of devices, including some under a consumer's control that could be particularly attractive tools for attacking meters.

Many networks of corporate desktop clients offer high-speed connectivity between clients and local servers and are centrally-administered. They are well-suited to XIVE when they also enforce a policy regarding which programs clients are permitted to run. They may face a variety of physical threats, but the primary vulnerabilities and threats affecting desktop clients are generated directly or indirectly by their users operating standard I/O devices. Users may run vulnerable applications that are subsequently attacked by remote entities, run malicious applications directly, or attempt to install legitimate applications that are disallowed by the policy for non-security reasons.

4.4 Implementation

We synthesized IAP to run on the Digilent XUPv5 development board. It is based on version 4104 of the LEON3. We also ported the changes in IAP that improve performance in standard software to a reference version of the LEON3, which we used as the basis for a series of benchmarks. Specifically, these enhancements include support for selective I-TLB and I-cache flushing and a slight modification to the DDR memory synchronization buffers. Both versions of the processor have the following configuration features: D-cache with 1 set of 8KiB and a line size of 4 words, D-cache system bus snooping, D-TLB with 8 entries, I-cache with 4 sets of 32KiB each and a line size of 8 words, I-TLB with 64 entries, hardware support for multiplication and division, branch prediction, dual JTAG and serial interfaces for debugging, 100Mbps full-duplex Ethernet, and 256MiB of DDR2 main memory. Each of the TLB and cache structures in both processors implements a Least Recently Used (LRU) replacement policy. We synthesized both processors at a clock frequency of 50MHz using the Xilinx Synthesis Tool (XST) v.13.1.

FPGAs are implemented differently than commercial processors, so we are unable to provide a simple comparison of the silicon area and quantity of wire used by the two implementations. Instead, we compare the quantity of FPGA resources that are utilized. The reference utilizes 52% of the slices and 33% of the BlockRAMs and FIFOs. IAP utilizes 71% of the slices and 40% of the BlockRAMs and

FIFOs. Note that the additional silicon used by IAP for cryptographic acceleration will remain dark for substantial periods of time. Recent trends in processor designs suggest that dark silicon space will become increasingly plentiful [67].

The prototype permits access to the coprocessor registers by the target, so that the target can initiate the integrity kernel ROM load process. It does this by installing a pattern for a software trap, and then triggers that trap to initially enter integrity kernel mode. A deployable implementation of IAP would disable that access as well as the debugging interfaces. Another debugging feature in our prototype that would not be present in a deployable implementation is that the integrity kernel ROM data is actually stored on the auxiliary RAM at the time IAP begins executing, before being copied into the on-chip integrity kernel space. This permits easy experimentation with various versions of the integrity kernel. Code outside the IK can access some BLAKE and AES functionality in our prototype, which would be disallowed or carefully controlled in a deployable implementation. Finally, the hash cache described in §4.5 is also located in auxiliary RAM, leaving it vulnerable to modification in the prototype. A deployable implementation would expand the integrity kernel RAM and store the hash cache within it.

4.5 Evaluation

4.5.1 TCB Size

XIVE was constructed to demonstrate that a very small integrity kernel is capable of enforcing eXecuting $\rightarrow$ Verified on IAP. We compare the size of XIVE against that of other systems with similar objectives in Table 4.1. All are discussed in §5. XIVE is clearly much smaller than Patagonix, due to the fact that Patagonix is incorporated into the full-featured Xen hypervisor. Like XIVE, SecVisor was developed with the specific objective of minimizing its size, so it is much closer. To be fair, we calculated the code size of SecVisor from the breakdown of code they provided, excluding their SHA-1 and module relocation implementations since XIVE does not contain analogous software functionality. SecVisor must use page tables to detect the execution of unverified code and to protect itself, which introduces additional complexity compared to XIVE. We thank the authors of [10] for furnishing us with the relevant line count in Table 4.1, which includes comments and debugging code, and could perhaps be reduced by future optimizations.

System	Lines of Code
XIVE	932
Patagonix	3544 + ~230K (Xen)
SecVisor	2682
HyperSentry	~3400

Table 4.1: Comparison of the TCB sizes of various systems.

4.5.2 Performance Methodology

We evaluated the performance implications of XIVE using a series of benchmarks. We developed a Python script to automatically run the benchmarks and record the time consumed by each test. It controls and monitors the benchmarks using the target's Linux serial console and the serial debugging interface. The whole series of tests was run in sequence ten times for each processor configuration.

The reference system retains the network driver receive buffer handling adaptation described in §4.2.9. It introduces substantial overhead in network processing, but we retain it since we want to highlight the overhead introduced by XIVE's network traffic. It may be possible to optimize the network driver in the future to reduce its overhead.

Figure 4.4a shows results from testing XIVE in two configurations. The one labeled "Hash Cache" includes the full functionality of the XIVE kernel as described in §4.2.9. The one labeled "No Hash Cache" disables the hash caching functionality, since it is not obvious a priori which configuration imposes less overhead.

Additionally, to demonstrate the overhead inherent in software-based approaches, we modified the Linux kernel to use page table manipulations to trap attempts to execute unverified code. The kernel does not actually hash or verify the code, so most of the overhead is generated by the page table manipulations and associated traps themselves. We compared the results of benchmarks running that configuration on the reference hardware, labeled "Page Tables," against a third configuration of XIVE that does not perform any hashing or network communication, labeled "Trap Only," in Figure 4.4b.

Each of the configurations just discussed was used to run a series of five tests. The first test ("Create Processes") is a microbenchmark that demonstrates process creation overhead in an adverse case. It executes the `ls` command in an empty directory 10 times in succession. Since `ls` is a lightweight command that performs little work in this case, it demonstrates the time that XIVE requires to verify code during process creation and destruction.

Second, we tested the time it takes the Linux kernel to boot ("Boot Kernel"). We modified the kernel to print out the base name of the command for each process at the time that it is launched, so we considered the kernel to be fully booted when we detected that the `init` process had been launched.

Third, we downloaded a 2MiB file using the `wget` command from an instance of the LigHTTPD server running on the approver machine ("Download HTTP"). This test demonstrates that XIVE is capable of sharing the Ethernet interface with the target OS.

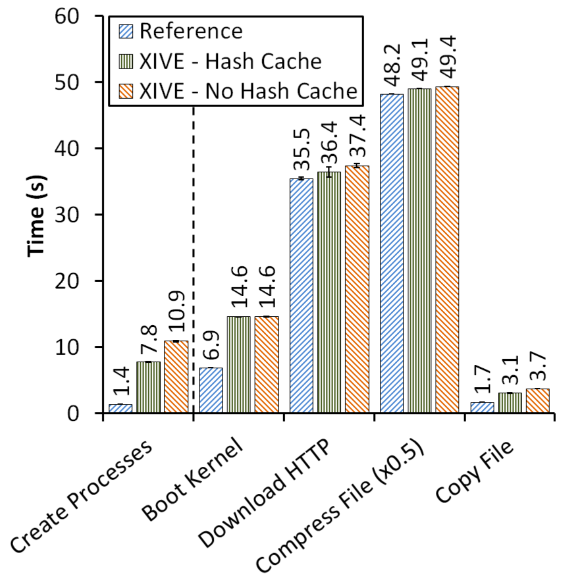
Finally, we demonstrated the effect of XIVE on a computationally-expensive process by compressing the previously-downloaded file using GZip ("Compress File"). GZip ordinarily involves a mixture of IO and computational operations, but the entire filesystem of our prototype is hosted in RAM, so IO is relatively fast. This fact can also be derived by noting the time difference between this test and the next one showing the cost to copy the same file ("Copy File"). We scaled all results from the compression test by a factor of 0.5 to prevent them from dominating the chart.

The FPGA was directly connected with an Ethernet cable to the machine running the approver. That machine was a Thinkpad T61 with a 2GHz Core 2 Duo processor and 2GiB of RAM running Ubuntu 10.10 64-bit desktop edition.

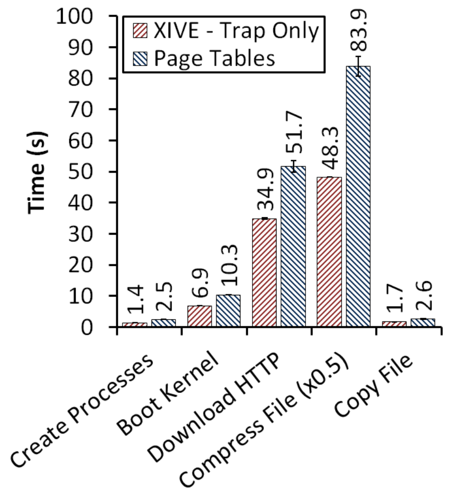
4.5.3 Performance Results

In general, the benchmark results in Figure 4.4a demonstrate that XIVE imposes low overhead for important types of tasks. However, we present a detailed explanation of the "Create Processes" microbenchmark results as well as those of the "Boot Kernel" benchmark below.

We hypothesized that the repeated verification of shared kernel code and userspace libraries was partially responsible for the order of magnitude performance degradation observed between the reference and the "No Hash Cache" configuration, which is what prompted us to develop the hash caching scheme. It is apparent from the results generated by the "Hash Cache" configuration that caching hashes for resident pages of memory dramatically reduces process creation and destruction overhead. It is also apparent that the overall effect of hash caching



(a) Reference configuration and enforcing configurations of XIVE. The test to the left of the dashed line is a microbenchmark, designed to demonstrate process creation and destruction overhead in an adverse case.



(b) Trap-only configuration of XIVE and page table-based software implementation.

Figure 4.4: Mean time required to perform benchmarks, including bars to show the standard error.

on the large-scale benchmarks is often positive.

Booting the kernel involves verifying a large quantity of code, including several verifications of the entire 16MiB system image prior to setting up fine-grained page tables, so boot time suffers a substantial slowdown. It is possible to modify the non-MMU V bits to operate with a finer granularity or to use a more sophisticated mapping structure to reduce this overhead, but that increases hardware complexity and seems unwarranted given the fact that this is a one-time cost per system reset.

We used the "Hash Cache" configuration to determine that XIVE generated an average of 3.7MiB of verification-related network traffic with a standard error of 7KiB as a result of running each series of tests and associated administrative commands.

4.6 Summary

IAP is a processor technology that is specifically designed to efficiently support integrity kernels. It provides high performance, hardware-enforced isolation, high compatibility with target systems and flexible invocation options to ensure visibility into the target system. We demonstrated the utility of IAP by developing XIVE, a code integrity enforcement service with a client component that fits entirely within IAP's protected space, containing 859 instructions. XIVE verifies all the code that ever executes on the target system against a network-hosted whitelist, even in the presence of DMA-capable attackers.

Chapter 5

Related Work

This chapter discusses related work concerning foreign code detection (and prevention, in some cases) and other areas that are foundational to our work or related in some other way. Most foreign code detection mechanisms use code identity as the basis for their monitoring and decisions, as do the integrity kernels described in this dissertation. However, they differ greatly in the mechanisms they use as the basis for their security guarantees and performance characteristics. Thus, we categorize each work according to the architectural approach that best characterizes it, although some of the works actually use elements from multiple architectures.[1]

5.1 Coprocessor-Based Foreign Code Detection

The Linux Integrity Measurement Architecture supports remote attestation of a Linux system. It uses a TPM to record the configuration of the system and to provide a signed copy of that configuration information to authorized remote challengers [58]. It only maintains information about the configuration of a system since it was last reset.

The Mobile Trusted Module standard supports the remote attestation features provided by the TPM, as well as secure boot functionality [69]. It also supports a small implementation footprint and non-ASIC implementations, including software implementations [73].

It is possible to use a TPM with a sensor node to support remote attestation and other services [33]. However, that paper does not discuss how to securely handle remote firmware upgrades, which is substantially more challenging than performing remote attestation of a static firmware image.

Attested Append-only Memory maintains a cumulative record of logged kernel events in an isolated component to increase the proportion of attackers that can

[1]This chapter includes material from previous publications by LeMay and Gunter [43, 44].

be safely tolerated within Byzantine-fault-tolerant replicated state machines [19]. Their architecture proposals are oriented towards server applications, but the paper provides examples of how attested information besides application firmware identity can be useful. The Trusted Incrementer project showed that the trusted computing base for Attested Append-only Memory and many other interesting systems can be reduced to a simple set of counters, cryptography, and an attestation-based API implemented in a trusted hardware component [46]. A CAK could be adapted to provide similar functionality in firmware with a potentially different threat model.

5.2 Software-Based Foreign Code Detection and Remote Sensor Node Recovery

DataGuard detects when a data object is overflowed by first causing the application to initialize canary values surrounding data objects using seed data that is deleted from the sensor node immediately after initialization, and then permitting the verifier to perform remote attestation by demanding that the sensor node produce the canary values [74]. Since the seed data is no longer available to the sensor node at the time of attestation, a compromised node that was infected using a buffer overflow will be unable to produce the canary values. However, this assumes that the application is initially trusted to not store the seed data, which is a weaker threat model than the one we apply to CRAESI and CESIum.

As an alternative to attesting the code currently installed on a sensor node, it is also possible for a sensor node to prove to a remote verifier that it has securely erased all of its code, which can then be used as a foundation for proving that it has subsequently installed specific new code [56]. This approach requires that the code to coordinate that process be installed in ROM. It also requires that the sensor node not offload computation during attestation, making it more narrowly applicable than CRAESI and CESIum.

Attempts to perform return-oriented programming can be detected with high probability by requiring all return addresses to be stored redundantly in encrypted form on the stack [51]. However, it is not necessary to use such a complex scheme in CESIum, as we have demonstrated. Furthermore, it is necessary to consider several complex ways in which attackers can manipulate kernel code to bypass CFI enforcement and potentially steal kernel secrets that can be leveraged in future attacks.

SWATT is an approach to verify the memory contents of embedded systems [62]. Its basic operating model assumes that the external verifier knows the precise type of hardware installed in the embedded system to be verified, that the network exhibit low jitter, and that the system being verified not be able to offload computation to an external device. Embedded systems often operate on networks where the latter two assumptions is not valid. It provides no intrinsic assurances of the continuous proper operation of embedded systems. Other potential pitfalls have been identified for attestation approaches on embedded systems that involve software, and CRAESI and CESIum both avoid those pitfalls [18].

CRAESI and CESIum do not explicitly attempt to prevent embedded system compromise or provide any mechanism for securely deploying firmware updates. They only allow remote verifiers to detect the presence of untrusted firmware during the past execution of the system. However, recovering from compromises can be expensive, so it is critical that compromises be prevented whenever possible, using good coding practices and any other applicable techniques, and that a secure update mechanism be used to deploy firmware updates.

SCUBA is a software-based system for recovering sensor nodes that have been compromised with malicious firmware [59]. It is based upon a revised version of the Pioneer primitive [61], and uses self-checksumming code to construct an indisputable code execution environment, which allows a remote party to ensure that a specific code image is atomically executed on a remote sensor node. In SCUBA, the particular code image that is used has the sole purpose of installing a firmware image that is provided by the verifier. Of course, the malicious firmware on the sensor node may interfere with this update process, in which case the node must be blacklisted and manually restored later. This scheme can guarantee the atomic completion of the restoration operation, but it does not provide any assurances about the past or future operation of the node and exhibits many of the same limitations as SWATT. Additionally, it requires the attacker's hardware to not be present in the network at the time of the restoration process. However, if all these assumptions can be satisfied in particular systems, SCUBA provides a useful technique for remotely restoring a compromised node.

Sluice uses a progressive verification scheme to efficiently propagate updates in a secure manner by constructing "pipelines" of nodes that sequentially propagate small portions of updates after individually verifying their origin and integrity [41]. No updates are applied until being verified, which helps to prevent some batterystealing attacks that exploit the energy-intensive nature of flash memory updates.

5.3 Hypervisor-Based Foreign Code Detection

SecVisor seeks to ensure that all kernel code ever executed is approved according to a user-defined policy [60]. The prototype uses a whitelist of hashes as the policy. SecVisor uses a variety of mechanisms to enforce the policy, all based on an underlying hypervisor. XIVE monitors all code executed on the target system, including userspace.

HyperSafe focuses on protecting the hypervisor itself by tightly controlling access to page tables and enforcing control flow integrity [72]. The hypervisor must be instrumented to guarantee control flow integrity. They noted that instrumenting their prototype required some manual analysis.

Patagonix uses a modified Xen hypervisor and a management VM to measure code executed in a guest VM [47]. It relies only on MMU and hypervisor protections to detect code execution. It uses identity oracles to identify programs and then reports them to administrators. The identity oracles are constructed to efficiently identify programs and handle relocated code. Patagonix handles JITs by excluding the code that they produce from analysis. Patagonix has a large TCB, comprising the hypervisor and the management VM. XIVE supports self-modifying code, unlike Patagonix.

Lares isolates the integrity kernel by using the Xen hypervisor to run the service in the system management VM, although they claim that Lares can be extended to place the service within a dedicated VM [55]. The integrity kernel monitors other VMs, but not Xen itself. The service can insert hooks at arbitrary locations within each target VM that then transfer control to the service. The main challenge faced by the author of a Lares integrity kernel is determining the proper locations for hooks to achieve comprehensive protection. IAP allows integrity kernel authors to more directly target security-relevant functionality.

Hypervisor-Based Integrity Measurement Agent measures and enforces the integrity of user programs running in target VMs on Xen by intercepting security-relevant events such as system calls leading to process creation in the target and by permitting only measured pages to execute [9]. It assumes that the hypervisor is statically measured, and it is tolerant of compromised kernels in VMs.

TrustVisor creates small VMs that isolate individual functions from a larger overall system and persists their state using TPM-based sealed storage [50]. XIVE monitors and controls the configuration of the whole system, which is largely an orthogonal concern.

5.4 Foreign Code Detection Using Processor Security Extensions

First, we provide background on SMM, since it is used by two of the papers in this section. SMM is intended to handle system events such as memory and chipset errors or processor thermal events. A System Management Interrupt (SMI) can be triggered by sending an electrical signals to a processor pin and by sending output to special IO locations [24]. When the processor enters SMM in response to an SMI, it jumps to a location in physical memory that is based on the value of an internal configuration register. That location is within a region of memory called System-Management RAM (SMRAM). Physically, SMRAM is located in main memory, but the chipset prevents access to it except by the processor in SMM or when the chipset's internal configuration registers are set to particular values. Kernel-level malware can read and overwrite SMM handlers on many older systems. The malware can replace the SMM handler in the cache without modifying the original SMM handler in main memory, which defeats integrity-monitoring mechanisms that monitor that space. Attackers accomplish this primarily by cleverly manipulating processor configuration registers that control caching.

HyperSentry and HyperCheck both use SMM to isolate an integrity kernel while it monitors the integrity of a hypervisor [10, 71]. HyperSentry uses an external management device called an Intelligent Platform Management Interface to permit arbitrary remote invocations of the integrity kernel by generating SMIs. They overcome limitations of SMM that make it difficult to reliably retrieve certain processor state, such as that associated with the Intel hardware virtualization extensions. HyperCheck offloads a significant amount of functionality to a PCI Network Interface Card (NIC). It uses the DMA capabilities of the NIC to directly read critical hypervisor code and data and send those to a network host that evaluates the snapshots for deviations from a baseline state. They rely on the SMM module to read processor registers and verify that they change in approved ways and to translate virtual addresses to the physical addresses required by the NIC. Both systems exhibit TOCTTOU vulnerabilities, due to their periodic nature, and also depend on comprehensively identifying security-critical code and structures. However, they do have the advantage of measuring program data as well as instructions.

The Cell Broadband Engine Isolation Loader permits signed and encrypted applications to be loaded into the Synergistic Processing Elements in the Cell processor [53]. It uses a layered architecture based in hardware. The hardware authenticates and loads a system software layer that then decrypts and authenticates a user-provided application against certificates and hashes included in the application assembly itself. The software loader erases itself before transferring control to the user application, to protect its secrets. Unlike XIVE, this architecture does not perform any ongoing monitoring of the executed code.

ARM TrustZone is a collection of hardware features that isolate the "secure world" from the "normal world" [8]. TrustZone-enabled processors implement 2 virtual cores, one for each world. System bus accesses are tagged according to the world from which they originate, and bus slaves enforce security restrictions based upon that tag. Transitions from the normal world to the secure world can be triggered by hardware interrupts or a special instruction. TrustZone does not include support for directly detecting attempts to execute unverified code. Intel Trusted Execution Technology and AMD Secure Virtual Machine have similar characteristics [35, 3].

5.5 Other Work

The Capability Hardware Enhanced RISC Instructions (CHERI) project is an example of another way in which processor hardware can be extended to help isolate integrity kernels and other system components [54]. The CHERI processor is based on a standard RISC instruction set, but includes special registers, instructions, and memory tags that support capabilities. This provides compatibility with legacy software and the ability to incrementally apply capability-based protections to portions of the target system. Capabilities contain type information about the resources that they protect, and they can operate at a finer granularity than memory pages.

NOVA is another interesting approach to isolation that is based on virtualization and separates the hypervisor from the virtual machine monitor, placing only the hypervisor in privileged kernel space [66]. The hypervisor comprises only 9000 lines of code, and only provides functionality for communication, resource delegation, interrupt control, and exception handling. It is distinguished from microkernels by its focus on full virtualization, instead of paravirtualization.

The ReVirt project has shown that it is feasible to maintain information on the execution of a fully-featured desktop or server system running within a virtual machine that is sufficient to replay the exact instruction sequence executed by the system prior to some failure that must be debugged [25]. DejaView uses a kernel-level approach to process recording to allow desktop sessions to be searched and restarted at arbitrary points [40]. It is conceivable that these techniques could support a CAK for desktops and servers, although it may not be feasible to store cumulative information for a long enough period of the system's life to be useful.

One primary factor leading to the security issues in hardware security coprocessors is the complexity of their APIs [32]. To ease analysis and reduce the incidence of vulnerabilities our designs export very simple APIs. We have analyzed the security of CRAESI using a model checker.

A previous methodology for modeling faults that can occur in systems and verifying that the systems tolerate those faults using a model checker only gives examples of logical faults, such as dropped messages [12]. We analyze the tolerance of CRAESI against physical faults, such as power supply interruptions.

Chapter 6

Conclusions and Future Work

The goal of this dissertation was to demonstrate that it is possible to develop compact integrity kernels to protect commodity microcontrollers in remote sensor networks, and that a custom hardware architecture can enable the construction of compact integrity kernels with superior isolation, visibility, performance, and compatibility. This shows that modifying hardware to more effectively support integrity kernels is feasible and helpful. It also provides a specific example of such modifications, although many other types of enhancements would undoubtedly be interesting as well. We have accomplished this demonstration in five main phases.

First, we argued that it is important to provide remote attestation support for remote sensor networks by using AMI as an example. Advanced meters support remote firmware upgrades, and that functionality could be abused to install malware. Advanced meters may also have other vulnerabilities that could be exploited to similar ends. Large-scale attacks on AMI could cause large-scale physical effects on the electric power grid such as blackouts. Existing remote attestation mechanisms do not support remote sensor networks that exhibit high jitter and permit sensors to collude with other devices, and that permit the firmware of sensors to be remotely upgraded.

We then argued that existing processors do not adequately support integrity kernels. Existing processors incorporate various security mechanisms that have been used to implement integrity kernels, but those kernels exhibit limitations that stem at least partially from the inadequacies of the underlying hardware security mechanisms. Such limitations are apparent in the CRAESI and CESIum integrity kernels, as well as integrity kernels from other authors. A major theme of our work on CRAESI and CESIum is the ability to create integrity kernels despite the limitations of the underlying hardware. This reflects the process of discovering and overcoming hardware-related challenges that we experienced while creating them. Chronologically, we first attempted to create an integrity kernel for an 8-bit Atmel AVR processor, since that was a good choice for inclusion in advanced meters at the time. We summarized the challenges involved in creating an integrity kernel for such processors in our discussion of CESIum. We then investigated the newer AVR32 family of processors that was also well-suited for use in advanced meters. That processor family provided additional memory protection capabilities in hardware that made it easier to construct an integrity kernel, and even allowed us to implement CRAESI without the support of a coprocessor and external flash memory. However, the processor still lacked useful security functionality, such as specific support for isolating an integrity kernel and detecting attempts to execute unverified memory. This motivated us to develop IAP.

Third, we presented the details of CRAESI, which is an integrity kernel for remote sensor nodes based on commodity flash MCUs that include a moderate amount of flash memory and an MPU. That integrity kernel includes code to protect itself from the target system by manipulating the MPU, and to block all write accesses by the target system to the flash memory containing the program code. This ensures that CRAESI is able to audit all code that is ever executed by the application. It uses software implementations of ECC and SHA-1 to implement remote attestation. The kernel consumes 81,312 bytes of program memory. We reserved 88KiB of flash memory to store the kernel code, and another 40KiB to store the persistent data manipulated by the kernel, out of a total of 512KiB of flash memory built into the MCU. We set aside 12KiB of data RAM for the kernel. Since CRAESI is implemented in software, it does not consume any energy while idle, compared to the TPM's idle power demand of 10.6mW. We used the Maude model checker to formally prove with respect to a reasonable model that CRAESI correctly implements firmware auditing and that its flash memory manipulations correctly tolerate repeated, unexpected power supply interruptions, assuming the interruptions eventually cease.

Next, we discussed how CRAESI can be adapted for commodity flash MCUs that have small flash memories or lack an MPU, resulting in the CESIum system. This can be accomplished by introducing an external flash memory and a second microcontroller that implements much of the functionality of the integrity kernel. It communicates over a local bus with the integrity kernel on the main microcontroller that contains the target system, encrypting that communication using AES-CCM. The only integrity kernel functionality that is hosted on the main microcontroller is the software required to communicate with the coprocessor, to implement the firmware upgrade process for the target system, and to protect the integrity kernel from attacks launched by the target system, particularly control-flow attacks. The total firmware image running on the CAC requires 24,346 bytes of flash program memory and 820 bytes of EEPROM. The integrity kernel requires 8,090 bytes of program flash memory out of 8,192 bytes available, and one byte of EEPROM. The kernel only uses data RAM while it is active, so it does not restrict the applications data RAM usage in any way. The CAC demands 1.6mW when idle, which is much less than the TPM's power demand.

Finally, we presented IAP, a SPARC processor that had been extended with special support for integrity kernels. It provides intrinsic isolation guarantees for integrity kernels by placing them in a dedicated address space and preventing the target system from initiating transfers to or from that space. It also includes special support for detecting attempts to execute unverified memory, even in the presence of attackers with DMA capabilities. We developed the XIVE integrity kernel for IAP that checks all unverified code against a network-hosted whitelist before permitting it to execute. The kernel contains only 859 instructions. We compared it against integrity kernels with similar objectives and showed that it is the most compact. We evaluated its performance and showed that its fundamental performance characteristics are superior to those of past techniques.

XIVE is currently unable to effectively handle bytecode and JITs, so we have proposed future directions that will resolve that shortcoming. It is also unable to completely prevent all attacks based on return-oriented programming, and we have discussed how that can be accomplished. Another promising direction is exploring how other architectures besides SPARC can be enhanced to serve as IAPs. We also plan to explore the requirements for integrity kernels in health information technology.

References

[1] Guidelines for smart grid cyber security. National Institute of Standards and Technology IR 7628, August 2010.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, Alexandria, VA, USA, November 2005.

[3] Advanced Micro Devices. AMD64 architecture programmers manual, volume 2: System programming. *Publication Number: 24593*, June 2010.

[4] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. In *Proceedings of the 19th IFIP Working Group 11.3 Working Conference on Data and Applications Security*. Storrs, CT, USA, August 2005.

[5] Ross Anderson and Shailendra Fuloria. On the security economics of electricity metering. In *Proceedings of the 9th Workshop on the Economics of Information Security*, WEIS '10, Cambridge, MA, USA, June 2010.

[6] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136. Paris, France, April 1997.

[7] June Andronick, David Greenaway, and Kevin Elphinstone. Towards proving security in the presence of large untrusted components. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV '10, Vancouver, BC, Canada, October 2010.

[8] ARM Limited. ARM security technology—Building a secure system using TrustZone technology. *PRD29-GENC-009492C*, April 2009.

[9] Ahmed M. Azab, Peng Ning, Emre C. Sezer, and Xiaolan Zhang. HIMA: A hypervisor-based integrity measurement agent. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 461–470, Honolulu, HI, USA, December 2009.

[10] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 38–49, Chicago, IL, USA, October 2010.

[11] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX mode of operation. In *Proceedings of the 11th IACR Workshop on Fast Software Encryption*, FSE '04, pages 389–407, Delhi, India, February 2004.

[12] Cinzia Bernardeschi, Alessandro Fantechi, and Stefania Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability*, 12(4):251–275, December 2002.

[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, Security '03, Washington, DC, USA, August 2003.

[14] Ken J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, The MITRE Corporation, Bedford, MA, USA, April 1977.

[15] Severin Borenstein, Michael Jaske, and Arthur Rosenfeld. Dynamic pricing, advanced metering and demand response in electricity markets. *Center for the Study of Energy Markets, University of California Energy Institute, UC Berkeley*, October 2002.

[16] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, Alexandria, VA, USA, October 2008.

[17] B.A. Carreras, V.E. Lynch, I. Dobson, and D.E. Newman. Critical points and transitions in an electric power transmission model for cascading failure blackouts. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 12:985–994, December 2002.

[18] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, Chicago, IL, USA, November 2009.

[19] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 189–204, Stevenson, WA, USA, October 2007.

[20] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 8th IEEE Symposium on Security and Privacy*, Oakland '87, Oakland, CA, USA, April 1987.

[21] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.1). *SRI International*, April 2005.

[22] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Cloaker: Hardware supported rootkit concealment. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, Oakland '08, pages 296–310, Oakland, CA, USA, May 2008.

[23] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[24] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the SMRAM: SMM reloaded. In *CanSecWest '09*. Vancouver, Canada, March 2009.

[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 211–224, Boston, MA, USA, December 2002.

[26] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. National Institute of Standards and Technology Special Publication 800-38C, May 2004.

[27] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In *Proceedings of the 4th International Workshop on Rewriting Logic and its Applications*, WRLA '02, pages 162 – 187, Pisa, Italy, September 2002.

[28] EPRI IntelliGrid Consortium. Automatic meter reading (AMR) and related customer service functions. `http://www.intelligrid.info/IntelliGrid_Architecture/Use_Cases/CS_AMR_Use_Cases.htm`, 2004.

[29] Hector J. Altuve Ferrer and Edmund O. Schweitzer, 3rd. *Modern Solutions for Protection, Control and Monitoring of Electric Power Systems*. Schweitzer Engineering Laboratories, Incorporated, 2010.

[30] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, Alexandria, VA, USA, October 2008.

[31] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Proceedings of the 3rd Workshop on Cryptographic Hardware and Embedded Systems*, CHES '01, pages 251–261. Paris, France, May 2001.

[32] Jonathan Herzog. Applying protocol analysis to security device interfaces. *IEEE Security and Privacy*, 4:84–87, July 2006.

[33] Wen Hu, Hailun Tan, Peter Corke, Wen Chan Shih, and Sanjay Jha. Toward trusted wireless sensor networks. *ACM Transactions on Sensor Networks*, 7:5:1–5:25, August 2010.

[34] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, Security '09, pages 383–398, Montreal, Canada, August 2009.

[35] Intel. Intel trusted execution technology software development guide. *Document Number: 315168-006*, December 2009.

[36] ECMA International. ECMA-335: common language infrastructure (CLI), June 2006.

[37] International Business Machines. IBM X-Force 2010 mid-year trend and risk report. `http://www.ibm.com/services/us/iss/xforce/trendreports/`, August 2010.

[38] Paul A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the 8th IEEE Symposium on Security and Privacy*, Oakland '87, Oakland, CA, USA, April 1987.

[39] Paul A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the 9th IEEE Symposium on Security and Privacy*, Oakland '88, pages 130–139, Oakland, CA, USA, April 1988.

[40] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. DejaView: a personal virtual computer recorder. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 279–292, Stevenson, WA, USA, October 2007.

[41] Patrick E. Lanigan, Rajeev Gandhi, and Priya Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, Lisboa, Portugal, July 2006.

[42] Michael LeMay, George Gross, Carl Gunter, and Sanjam Garg. Unified Architecture for Large-Scale Attested Metering. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, Waikoloa, HI, USA, January 2007.

[43] Michael LeMay and Carl A. Gunter. Cumulative Attestation Kernels for Embedded Systems. In *Proceedings of the 14th European Symposium on Research in Computer Security*, ESORICS '09, pages 655–670, Saint Malo, France, September 2009.

[44] Michael LeMay and Carl A. Gunter. Enforcing executing-implies-verified with the integrity-aware processor. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, Pittsburgh, PA, USA, June 2011.

[45] Michael LeMay, Rajesh Nelli, George Gross, and Carl A. Gunter. An integrated architecture for demand response communications and control. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, HICSS '08, Waikoloa, HI, USA, January 2008.

[46] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, pages 1–14, Boston, MA, USA, April 2009.

[47] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium*, Security '08, pages 243–258, San Jose, CA, USA, July 2008.

[48] An Liu and Peng Ning. TinyECC: Elliptic curve cryptography on TinyOS. *http://discovery.csc.ncsu.edu/software/TinyECC/*, February 2011.

[49] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8:3–30, January 1998.

[50] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 143–158, Oakland, CA, USA, May 2010.

[51] Stephen McLaughlin, Dmitry Podkuiko, Adam Delozier, Sergei Miadzvezhanka, and Patrick McDaniel. Embedded firmware diversity for smart electric meters. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Security*, HotSec '10, pages 1–8, Washington, DC, USA, August 2010.

[52] Mono. http://www.mono-project.com.

[53] Masana Murase, Kanna Shimizu, Wilfred Plouffe, and Masaharu Sakamoto. Effective implementation of the cell broadband engine(TM) isolation loader. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 303–313, Chicago, IL, USA, November 2009.

[54] Peter Neumann and Robert Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Proceedings of the 4th Annual Layered Assurance Workshop*, LAW '10, Austin, TX, USA, December 2010.

[55] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 233–247, Washington, DC, USA, 2008.

[56] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *Proceedings of the 15th European Symposium on Research in Computer Security*, ESORICS '10, pages 643–662. Athens, Greece, September 2010.

[57] Pike Research. Smart meter installations to reach 250 million worldwide by 2015. *http://www.pikeresearch.com/newsroom/smart-meter-installations-to-reach-250-million-worldwide-by-2015*, November 2009.

[58] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, Security '04, San Diego, CA, USA, August 2004.

[59] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM Workshop on Wireless Security*, WiSe '06, pages 85–94, Los Angeles, CA, USA, September 2006.

[60] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, Stevenson, WA, USA, October 2007.

[61] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 1–16, Brighton, United Kingdom, October 2005.

[62] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the 25th IEEE Symposium on Security and Privacy*, Oakland '04, pages 272–282, Oakland, CA, USA, May 2004.

[63] SHA-3 proposal BLAKE. `http://131002.net/blake/`.

[64] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, Alexandria, VA, USA, October 2007.

[65] SharpOS Development Blog. `http://sharpos.blogspot.com`.

[66] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, Paris, France, April 2010.

[67] S. Swanson and M.B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *Communications Magazine, IEEE*, 49(4):112–119, 2011.

[68] Trusted Computing Group, Incorporated. TCG specification architecture overview. *Trusted Computing Group*, August 2007.

[69] Trusted Computing Group, Incorporated. Mobile trusted module specification, version 1.0. *TCG Published*, April 2010.

[70] Scott A. Vanstone. Next generation security for wireless: Elliptic curve cryptography. *Computers & Security*, 22(5):412–415, July 2003.

[71] Jiang Wang, Angelos Stavrou, and Anup Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection*, RAID '10, pages 158–177, Ottawa, ON, Canada, September 2010.

[72] Zhi Wang and Xuxian Jiang. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland '10, Oakland, CA, USA, May 2010.

[73] Johannes Winter. Trusted computing building blocks for embedded Linux-based ARM TrustZone platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 21–30, Fairfax, VA, USA, October 2008.

[74] Dazhi Zhang and Donggang Liu. DataGuard: Dynamic data attestation in wireless sensor networks. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 261–270, Chicago, IL, USA, June 2010.