Cumulative Attestation Kernels for Embedded Systems

Michael LeMay and Carl A. Gunter, *Senior Member, IEEE*

***Abstract*—To mitigate the threat of malware intrusions on networked embedded systems, it is desirable to provide remote attestation assurances for them. Embedded systems have special limitations concerning cost, power efficiency, computation, and memory that influence how this goal can be achieved. Moreover, many types of applications require integrity guarantees for the system over an interval of time rather than just at a given instant. We propose a *Cumulative Attestation Kernel (CAK)* that addresses these concerns. We demonstrate the value of CAKs for Advanced Metering Infrastructure (AMI) and show how to implement a CAK in less than one quarter of the memory available on low end flash MCUs similar to those used in AMI deployments. Regarding this prototype, we present the first formal proof we are aware of that a system is tolerant to power supply interruptions. We also discuss how to provide cumulative attestation for devices with tighter memory constraints by offloading computation and storage onto a *Cumulative Attestation Coprocessor (CAC)*.**

***Index Terms*—Intrusion detection, meter reading, power system security, smart grids.**

I. Introduction

NETWORKED embedded systems are becoming increasingly widespread and important. 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, since it is necessary to add new features to them in the field [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 [2]. Advanced meters potentially enable attackers to induce large-scale effects. For example, coordinated attacks on demand response systems controlled via meters may result in blackouts, since the US power grid may operate with a delicate balance between generation and load [3].

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 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. We argue that tamper-resistant firmware auditing on advanced meters can serve as such a mechanism. We also argue that AMI system administrators can use firmware auditing to detect attacks and respond to them in such a way that they are prevented from inducing large-scale effects.

A desktop or mobile system can use a Trusted Platform Module (TPM) to protect and certify audit information concerning its configuration, using a process called remote attestation [4], [5]. Unfortunately, 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 short time interval, which is incompatible with the deployment model of embedded systems such as advanced meters, which operate unattended for extended periods of time.

In this paper, we describe an architecture for providing remote attestation for advanced meters, which should also serve as an example of how it could be provided on other similar types of embedded systems. The architecture is 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 kernel 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 other 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. Finally, flash MCUs operate at low clock frequencies, and may not offer protection features such as a memory management unit (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 oper-

Manuscript received April 20, 2011; revised September 09, 2011; accepted October 17, 2011. Date of current version May 21, 2012. This work was supported in part by DOE DE-OE0000097, NSF CNS 07-16626, NSF CNS 07-16421, NSF CNS 05-24695, ONR N00014-08-1-0248, NSF CNS 05-24516, DHS 2006-CS-001-000001, HHS 90TR0003-01, NSF CNS 09-64392, NSF CNS 09-17218, and grants from the MacArthur Foundation, Boeing Corporation, and Lockheed Martin Corporation. The work of M. LeMay was supported on an NDSEG fellowship from AFOSR for part of this work. Paper no. TSG-00155-2011.

The authors are with the Department of Computer Science, Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302 (e-mail: mdlemay2@illinois.edu; cgunter@illinois.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSG.2011.2174811

ations 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)* [6]. 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 is called *Cumulative Embedded System Integrity (CESIum)* and is based on a platform using a coprocessor in addition to the main processor, both of which are 8-bit Atmel AVRs.

We do not extensively discuss node recovery in this paper, 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.

Our contributions are as follows: 1) requirements and design for CAKs that are fault-tolerant and respect the constraints of advanced meters; 2) prototype standalone CAK called CRAESI that satisfies these requirements for mid-range flash MCUs; 3) prototype CAC called CESIum that is suitable for lower-end flash MCUs; and 4) formal proof that CRAESI has certain security and fault-tolerance properties, including the first formal proof we are aware of that a system is tolerant to power supply interruptions.

The rest of this paper is organized as follows. Section II contains additional background. Section III describes the threat model and requirements for a CAK. Section IV discusses a design that satisfies those requirements. Section V presents experimental results from CRAESI. We formally analyze important properties of CRAESI in Section VI. We describe and evaluate CESIum in Section VII. We discuss additional related work in Section VIII. Finally, we conclude in Section IX.

II. Background

A. 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 $\mathcal{S}$. A variety of protocols can be used to accomplish this. They usually involve at least two messages. The request $\mathcal{V} \xrightarrow{\langle \nu \rangle} \mathcal{S}$ contains a nonce ν used to ensure the freshness of the attestation results. The response $\mathcal{S} \xrightarrow{[\langle \nu, \sigma \rangle]_{\mathrm{RTM}\mathcal{S}}} \mathcal{V}$ is digitally signed by the root-of-trust for measurement (RTM) of $\mathcal{S}$ (RTM$\mathcal{S}$) to certify that it has not been tampered, and contains the nonce ν as well as a record of the system's state σ. Of course, this assumes that the system contains some RTM$\mathcal{S}$ 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 [4]. 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 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 $\mathcal{S}$ 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 $\mathcal{S}$ 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 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.

B. 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 [2]. A report by Pike Research states that more than 250 million meters will be deployed worldwide by the year 2015 [7]. 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, but did not address the need for cumulative attestation or provide a design suitable for use on practical flash MCUs [8].

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 [9].

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 [10]:

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.

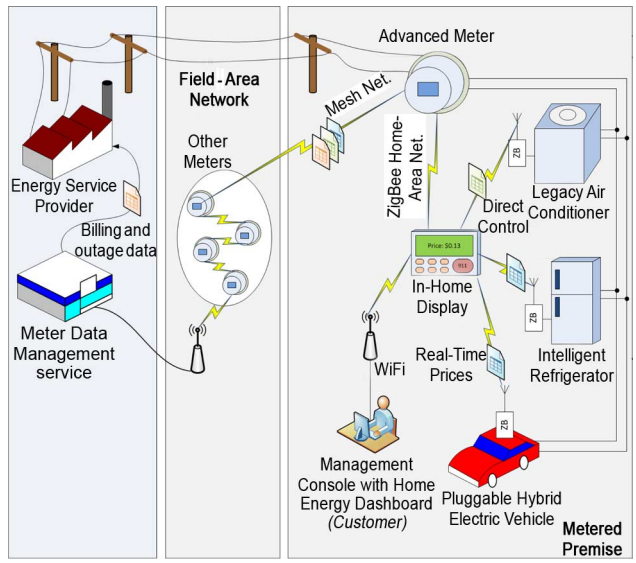

Fig. 1. Full-featured bidirectional metering network interactions.

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 service, and the most capable systems have fully bidirectional network connections with the service. We focus on meters with bidirectional connections in this paper. AMI networks with connectivity to the service can distribute real-time pricing schedules to meters, which can influence customer behavior and induce manual or automatic demand response actions [11]. They can also support direct control signals.

In Fig. 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 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. A bidirectional AMI network is useful for upgrading the firmware on meters to support HAN connectivity [12]. Firmware upgrades can also enable maintenance of meters and be used to introduce updated functionality. The frequency and total number of upgrades that will actually be issued is hard to estimate, since utilities can use AMI quite differently from each other. Some utilities may have ambitious demand response programs that make use of cutting-edge, frequently-updated HAN technology, whereas other utilities may only use basic meter-reading functionality that is rarely updated.

TABLE I
TEMPORAL MODAL OPERATORS USED IN LTL FORMULAS

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.

C. Formal Methods

Formal methods are used to verify correctness and fault-tolerance properties of CRAESI in Section VI. 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 [13]. Essentially, rewriting logic provides a convenient technique to express nondeterministic finite automata. Maude is a multiparadigm 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 $(\neg, \wedge, \vee, \longrightarrow)$ 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 I.

III. THREAT MODEL AND REQUIREMENTS

A. Threat Model

Data integrity on meters can be compromised by malicious application firmware in various ways, as shown in Fig. 2. 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) 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.

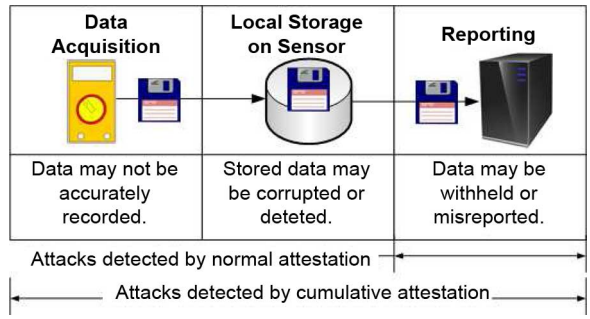

Fig. 2. Three modes of attack on sensor data available to malicious application firmware running during various lifetime phases occupied by that data.

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 [14], [15]. 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 [16]. 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 [17]. 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 [18].

B. 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: 3) *Cost-effectiveness:* Even the smallest added expenses in advanced meters become significant when multiplied for massive deployments. 4) *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. 5) *Suitability for hardware protections:* The CAK must be adapted to the protection mechanisms provided by the processor on which it runs.

IV. CAK Design

We now present a kernel design that satisfies the requirements. The basic flash memory layout of the system is depicted in Fig. 3. 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 nonexecutable data.

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

A. Audit Log

Each location in the audit log contains an entry $\epsilon_i = \langle \tau, \eta, \theta \rangle$, 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 $h_{\text{fold}}(x, y) = h(x \| y)$. $AL_{inmem} = AL$ when $|AL_{ovf}| = 0$ and $AL_{inmem} = (\epsilon_{chain}) + AL_{recent}$ otherwise. A counter $\lambda = |AL|$ is also included.

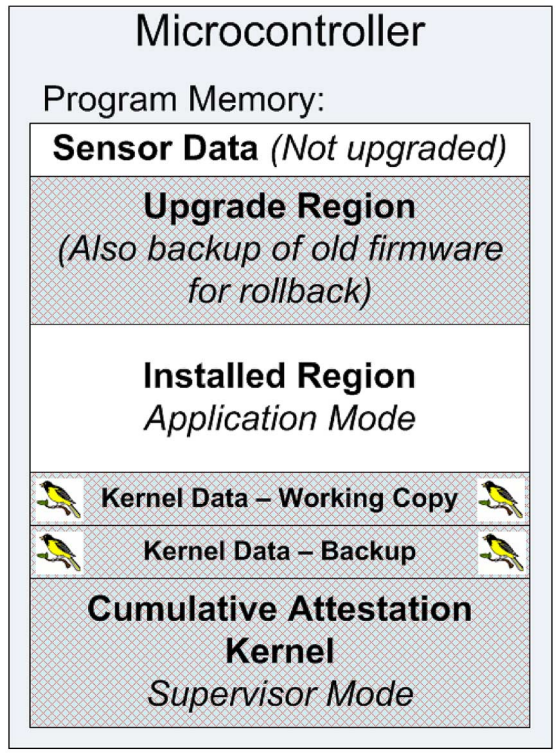

Fig. 3. The general CAK program memory layout. The birds represent "canary" values.

If AL has overflowed, the remote verifier must independently obtain the hash values that are no longer stored: $(h(\mathcal{F}_0), h(\mathcal{F}_1), \ldots h(\mathcal{F}_m))$. This is a reasonable assumption if the embedded system is used by a group of remote parties that know and can communicate with all parties that have legitimately installed new firmware revisions on the system. In that case, the remote verifier can use a separate protocol to request that the updaters provide whatever hash values the verifier does not yet know, so the verifier can construct a complete hash chain. For example, a shared repository at a well-known Internet location could be collaboratively maintained by all updaters and then accessed as needed by verifiers. An example of a simple and sufficient system is an FTP server containing a file for each upgrade specifying when it was installed, what its hash value was, and potentially other helpful metadata. The FTP server would need to only allow uploads from authorized updaters and only allow downloads from authorized verifiers. This would necessitate encryption of the FTP connections, some form of authentication, and the existence of an entity to administer the authentication system. A variety of approaches can be used to control which parties are authorized to update the system's firmware, such as controlling access to one or more private keys that are used to sign firmware updates. In this example, the keys could be distributed to additional parties at any time, even after the meter has been deployed. An attacker is likely to be unknown or to refuse to provide hashes of the firmware that it installed, but the past or current presence of firmware installed by the attacker will still be detected by the

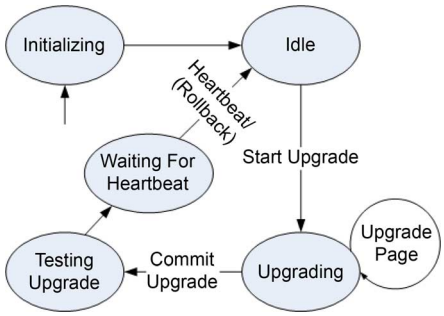

Fig. 4. A basic state machine representation of CAK operation, in which transitions are generated by the specified commands.

verifier when the list of known hashes does not produce the expected hash chain value.

B. Asymmetric Keypairs

The public and private keys for keypair P_x are denoted as Y_x and R_x, respectively. The firmware audit private key $R_{\mathcal{F}}$ is used to sign the firmware audit log during attestation operations. The Diffie-Hellman keypair P_{DH} is used during Diffie-Hellman key exchanges. The master private key R_Ω is used to sign the firmware audit public key $Y_{\mathcal{F}}$ and the Diffie-Hellman public key Y_{DH}. The keys can be generated by the CAK when it is first initialized using a random number generator (RNG) that we assume is available. 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.

C. System State

An explicit state variable σ is used to control transactions. States in the automata in Fig. 4 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 Fig. 4 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 Section VI. 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)
    γ0 ← False
    γ1 ← False
    Φw ← INSERT(addr, data, Φw)
    γ0 ← True
    γ1 ← True
end procedure
```

The copies of the filesystem have canary values γ_i before and after the file data as depicted in Fig. 3. Whenever a file in the working copy is modified, canaries γ_0 and γ_1 are first invalidated and then reinitialized 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
    γ2 ← False                          ▷ aborted(PostCritical)
                                        (¬aborted(PreCritical))
    γ3 ← False
    ⟨γ2, Φp, γ3⟩ ← ⟨γ0, Φw, γ1⟩
end procedure
```

The presence of comments in the pseudocode, like "$aborted(PostCritical)$ $(\neg aborted(PreCritical))$" 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 Section VI.

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 ⟨γ2, γ3⟩ ≠ ⟨True, True⟩ then
    Φp ← Φw
  else if Φw ≠ Φp then
    Φw ← Φp
  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.

Listing 4 Load Firmware Upgrade Into Executable Space

```
procedure UPGRADECOMMIT
  while σupg.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 σupg.stage = Staging then ▷ stage is initialized to
Staging when an upgrade is first initiated.
      codeStagingArea ← upgradeRegion_n
      σupg.stage ← BackingUp
      FSCOMMIT
    else if σupg.stage = BackingUp then
      upgradeRegion_n ← installedRegion_n
      σupg.stage ← Finishing
      FSCOMMIT
    else if σupg.stage = Finishing then
      installedRegion_n ← codeStagingArea
      σupg.n ← σupg.n + 1
      σupg.stage ← Staging
      FSCOMMIT
    end if
  end while
  σ ← TestingUpgrade
  FSCOMMIT
end procedure
```

Only the kernel has permission to actually write to the upgrade region. The application issues requests to the kernel to write individual pages and the kernel then performs the write operations. Thus, the application is responsible for properly initializing the upgrade region even in the presence of undesirable situations such as simultaneous upgrade requests from different remote parties. Regardless, the kernel will audit whatever firmware is actually installed.

Every time the meter boots, the processor immediately transfers control to 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 σ = Init ∨ σ = Idle
    DIRPREP (None, Idle)
  else if σ = Upgrading then
    DIRPREP (UpgradeAborted, Idle)
  else if σ = TestingUpgrade
    DIRPREP (None, WaitingForHB)
  else if σ = WaitingForHB
    DIRPREP (UpgradeHBFailed, Idle)
  end if
end procedure
procedure DIRPREP (η, σ+)
  σ ← σ+
  FSCOMMIT
  if η ≠ None ∨ |ALinmem| = 0 ∨ εn−1 ≠ ⟨η, h(Fn)⟩
    LOGEXTEND (⟨η, h(Fn)⟩)
    FSCOMMIT
  end if
  JUMPMAIN                                      ▷ appfw-active
end procedure
```

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

V. 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 Section IV, 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.

A. Hardware Components

Our prototype implementation comprises five distinct devices. The first is an Atmel ATSTK600 development kit containing an AT32UC3A0512 AVR32 microcontroller with a 3.3 V 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.

TABLE II
LINES OF C++ CODE IN EACH CRAESI KERNEL COMPONENT

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

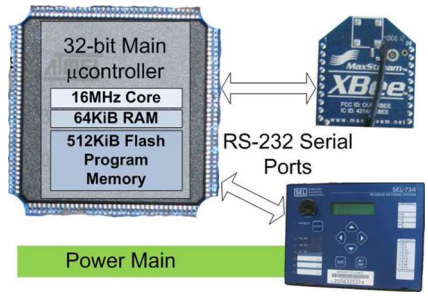

Fig. 5. Prototype hardware components and interconnects.

B. 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. Even a coprocessor-based approach would need to restrict access to security-critical resources, so this limitation is not specific to CRAESI.

C. 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) [19]. Pseudo-random numbers are generated by Mersenne Twister [20]. 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 RNG. Table II 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 88 KiB of flash memory to store the kernel code, and another 40 KiB to store the persistent data manipulated by the kernel. We set aside 12 KiB 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 191.5 KiB. 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.

D. 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 installed in a popular type of desktop PC. It was expensive and difficult to obtain measurements from this TPM, so we did not perform additional experiments on other TPMs. We used the TPM to perform similar operations to those that would be required by a TPM-assisted version of CRAESI if it were actually implemented. The TPM has a supply voltage of 3.3 V and relies on an LPC bus connection. We used digital multimeters (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 Fig. 6. 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 platform configuration registers (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 [21]. 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 10.6 mW 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 5 million

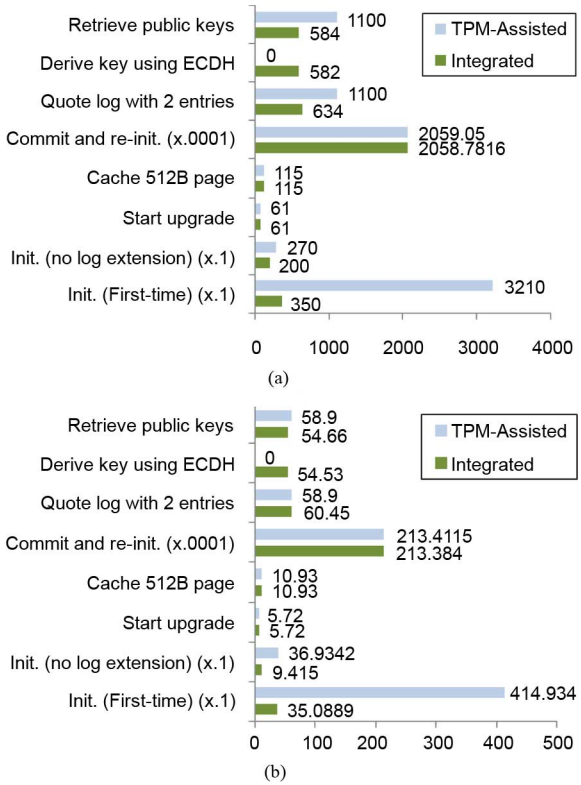

Fig. 6. A performance comparison of TPM-assisted and standalone (integrated) CRAESI. The energy measurements for TPM-assisted CRAESI represent the energy consumed by the TPM alone. (a) Elapsed time (ms). (b) Energy consumed (mJ).

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.

Another advantage of using ECC with a shorter key than the one used by RSA in a TPM is that less network data is generated by digital signature operations. An ECC signature using a 192-bit key over and accompanied by a 160-bit hash generates a total of 68 bytes of data, whereas the corresponding RSA signature using a 2048-bit key contains 256 bytes of signature data in addition to the 20 bytes of hash data, for a total of 276 bytes.

VI. 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 Section IV [22]. 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 [23]. We abandoned this approach when it became clear that a substantial development effort would be necessary to generate sufficiently efficient assembly code. LeMay's dissertation provides more details on our efforts to use C# [24].

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.

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 III.

The first theorem is concerned with the correctness and auditability of application firmware upgrade procedures. To express it, we stipulate that the system can occupy three primary states when the application is executing, as illustrated in Fig. 7. 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 Fig. 7 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

TABLE III
PROPOSITIONS USED IN LTL FORMULAS TO MODEL CHECK INTEGRATED CRAESI DESIGN

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.

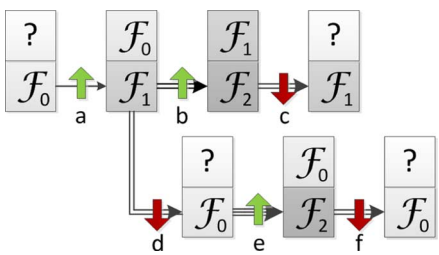

Fig. 7. Representative, legitimate firmware transitions.

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 Fig. 7; 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.*

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

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.*

$$\begin{aligned} &upgrade\text{-}inited(\mathcal{F}_1) \Rightarrow \\ &(\neg appfw\text{-}active\ \mathcal{U}\ ((installed(\mathcal{F}_1) \wedge cached(\mathcal{F}_0)) \\ &\mathcal{W}\ ((upgrade\text{-}started \wedge \bigcirc(installed(\mathcal{F}_1) \\ &\mathcal{W}\ upgrade\text{-}inited(\mathcal{F}_2))) \vee rollback))). \end{aligned}$$

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.*

$$\begin{aligned} &(installed(\mathcal{F}_n) \wedge upgrade\text{-}inited(\mathcal{F}_2)) \Rightarrow \\ &(\neg appfw\text{-}active\ \mathcal{U}\ ((installed(\mathcal{F}_2) \wedge cached(\mathcal{F}_n))\ \mathcal{W}\ rollback)). \end{aligned}$$

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.*

$$\begin{aligned} &(cached(\mathcal{F}_0) \wedge rollback) \Rightarrow (\neg appfw\text{-}active\ \mathcal{U} \\ &\qquad (installed(\mathcal{F}_0)\ \mathcal{W}\ upgrade\text{-}inited(\mathcal{F}_2))). \end{aligned}$$

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.*

$$\begin{aligned} &(cached(\mathcal{F}_1) \wedge rollback) \Rightarrow \\ &\qquad (\neg appfw\text{-}active\ \mathcal{U}\ (\Box installed(\mathcal{F}_1))). \end{aligned}$$

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.*

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

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 nondeterministic 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.*

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

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, nondeterministic 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 subsequently initialized if the transaction is interrupted prior to the critical point.*

$$(\neg\Box\Diamond rebooted \wedge \Diamond aborted(PreCritical)) \longrightarrow (\Diamond sff\text{-}inited \wedge (sff\text{-}inited \Rightarrow \Box sff\text{-}unchanged))$$

Lemma 8: *Executing on FS any filesystem transaction that is intended to update files according to $\varpi \subset FN \times FD$ results in $\varpi \cup \{(\nu, \delta) | (\nu, \delta) \in FS \wedge (\neg\exists\eta.(\nu, \eta) \in \varpi)\}$ 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.*

$$(\neg\Box\Diamond rebooted \wedge \neg\Diamond aborted(PreCritical)) \longrightarrow (\Diamond halted \wedge ((halted \vee sff\text{-}inited) \Rightarrow \Box sff\text{-}as\text{-}expected))$$

VII. 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.

A. 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 [25].

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 nonpreemptible. The CAC must not receive any command while executing a nonpreemptible 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 nonpreemptible commands, to ensure that attackers are unable to launch a time-consuming preemptible command that could then block a critical nonpreemptible 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.

B. Main Microcontroller

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.

Listing 6 Initialize Main Microcontroller Upon System Reset. Network Communications Between the Main Microcontroller $\mathcal{M}$ and the CAC $\mathcal{C}$ are Included

```
procedure BOOT
    M --init--> C
    M <--α-- C
    if α = newkey then                    ▷ The CAC needs to send a new key.
        M <--κ-- C
    end if
    υ ← ExternalFlash[UpgWaiting]
    for i ∈ (0, ..., |BuiltInProgMem|) do    ▷ Handle each page of program memory.
        π ← BuiltInProgMem[i]
        if υ = True then                  ▷ The application requested an upgrade.
            π_upg ← ExternalFlash[i]
            if π = π_upg then             ▷ The upgrade firmware page data matches the existing firmware page data.
                M --extend({π}_κ)--> C    ▷ Extend the CAC's firmware hash accumulator. Encrypt the page data.
            else
                BuiltInProgMem[i] ← π_upg    ▷ Upgrade the firmware page.
                M --extend({π_upg}_κ)--> C
            end if
        end if
    end for
    M --commit--> C                       ▷ Commit the firmware hash to the audit log if it is different from the latest entry.
end procedure
```

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.

C. Hardware Implementation

Our prototype comprises two interconnected circuit boards, which are depicted in Fig. 8. The board on the left is the Atmel ATSTK500 prototyping kit with an ATmega644V microcontroller, the CAC. The second board is the Atmel ATSTK600

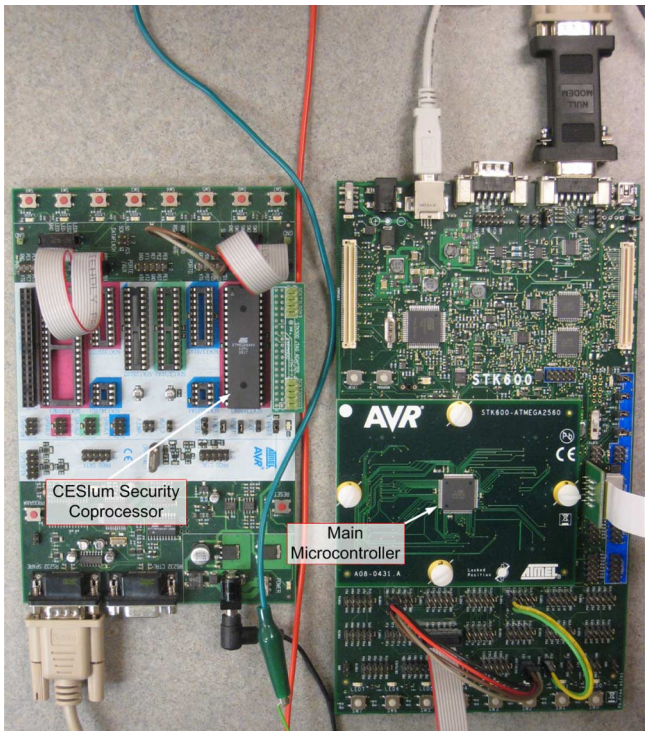

Fig. 8. Hardware prototype of CESIum.

prototyping kit with a daughtercard containing an ATmega2560 main microcontroller. The ATmega2560 has 256 KiB of program memory, but we only use 8 KiB for the kernel and 32 KiB 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 4 Mbit flash memory chip, which serves as the external flash memory.

D. Firmware Implementation

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.

1) 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 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 (ROP) so that the application regains control soon after the flash is written [18]. Since the AVR uses both 16- and 32-bit instructions with 16-bit address granularity, it is even possible that attackers could target ROP "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 [26]. 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 [27]. 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.

2) 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.

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

TABLE IV
LINES OF C CODE IN EACH COMPONENT

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

The kernel requires 8090 bytes of program flash memory out of 8192 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 IV 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.

E. Performance Evaluation

We used a similar experimental configuration to the one we used for evaluating the performance of CRAESI to evaluate the performance of CESIum. The processor was powered using a 2.0 V 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 1.6 mW when idle. The experimental results for various operations are presented in Fig. 9. 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 CESIum. 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 8 MHz system clock using an energy-efficient internal oscillator. The relative slowness of CESIum compared to the TPM is reflective of its 8-bit processing and slow clock rate.

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 32 KiB of firmware and performing the corresponding

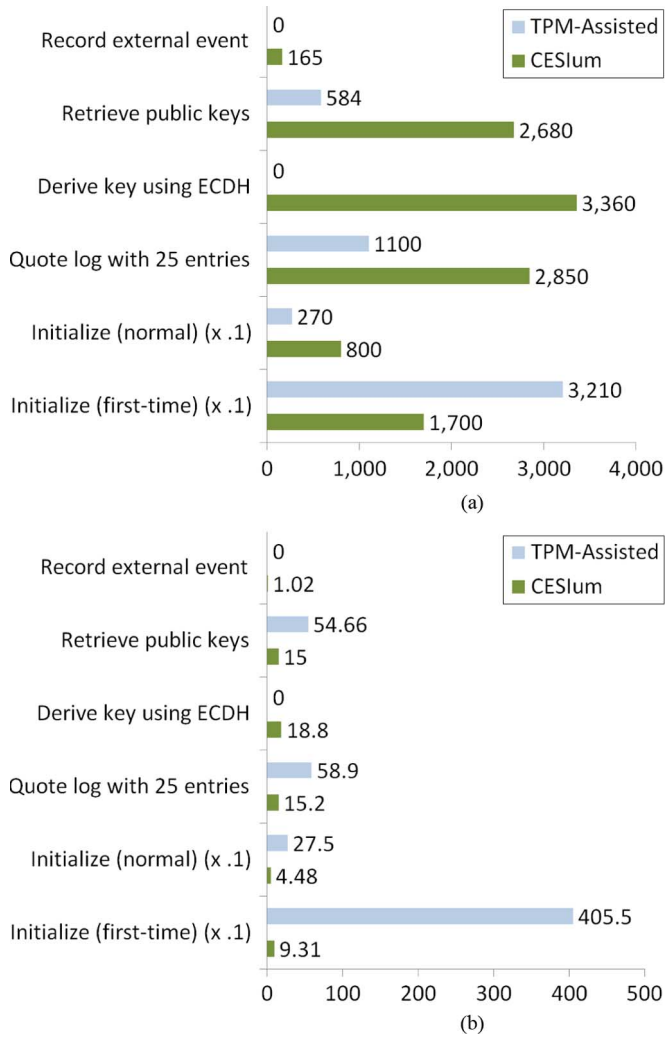

Fig. 9. A performance comparison of TPM-assisted and CESIum-based remote attestation. (a) Time (ms). (b) Energy (mJ).

log extension process for the initial firmware took an average of 25.4 s with a standard deviation of 0.5 s. The average time that the application required to initialize the external flash memory with 256 KiB of firmware and the associated command value in preparation for an upgrade was 6.2 s with a standard deviation of 0.8 s.

VIII. Related Work

The Linux Integrity Measurement Architecture (Linux-IMA) 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 [5]. 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 [28]. It also supports a small implementation footprint and non-ASIC implementations, including software implementations [29].

It is possible to use a TPM with a sensor node to support remote attestation and other services [30]. 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.

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 [31]. 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 [32]. 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 ROP can be detected with high probability by requiring all return addresses to be stored redundantly in encrypted form on the stack [33]. 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 [34]. 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 are 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 [35].

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 [36]. DejaView uses a kernel-level approach to process recording to allow desktop sessions to be searched and restarted at arbitrary points [37]. 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.

Attested append-only memory maintains a cumulative record of logged kernel events in an isolated component to increase the proportion of attackers that can be safely tolerated within Byzantine-fault-tolerant replicated state machines [38]. 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 [39]. A CAK could be adapted to provide similar functionality in firmware with a potentially different threat model.

SCUBA is a software-based system for recovering sensor nodes that have been compromised with malicious firmware [40]. It is based upon a revised version of the Pioneer primitive [41], and uses self-checksumming code to construct an indisputable code execution (ICE) 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.

Our attestation kernels 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. 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 [42]. No updates are applied until being verified, which helps to prevent some battery-stealing attacks that exploit the energy-intensive nature of flash memory updates.

One primary factor leading to the security issues in hardware CACs is the complexity of their APIs [43]. 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 [44]. We analyze the tolerance of CRAESI against physical faults, such as power supply interruptions.

Many operating systems have been formally verified to varying degrees [45]. The seL4 project is notable in that it resulted in verification of a microkernel down to the level of C source code, although the correctness of the compiler, assembly language code, and certain other components was assumed [46]. The total effort required to construct the proof was about 20 person-years, although the seL4-specific portions of that effort required only 11 person-years. The functionality to tolerate physical faults that is included in CRAESI has no analogue within seL4.

IX. Conclusion

We present 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. CESIum is suitable for use with low-end 8-bit flash MCUs.

Acknowledgment

Musab AlTurki assisted the authors in developing the formal model. The authors thank Ed Beroset, Samuel T. King, Sean W. Smith, Nabil Schear, Ellick Chan, the researchers in the Illinois Security Lab, the researchers in the TCIP and TCIPG research projects, and the anonymous reviewers for their feedback. The authors thank Schweitzer Engineering Laboratories for providing a substation meter that we used in our experiments. The authors measured lines of code using David A. Wheeler's "SLOCCount." The views expressed are those of the authors only.

References

[1] *Guidelines for Smart Grid Cyber Security, NIST IR 7628*. Gaithersburg, MD: National Institute of Standards and Technology, 2010.

[2] R. Anderson and S. Fuloria, "On the security economics of electricity metering," in *Proc. 9th Workshop Econ. Inf. Security, ser. WEIS '10*, Cambridge, MA, Jun. 2010.

[3] B. Carreras, V. Lynch, I. Dobson, and D. Newman, "Critical points and transitions in an electric power transmission model for cascading failure blackouts," *Chaos*, vol. 12, pp. 985–994, Dec. 2002.

[4] "TCG specification architecture overview," Trusted Computing Group, 2007.

[5] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proc. 13th USENIX Security Symp., ser. Security '04*, San Diego, CA, Aug. 2004.

[6] M. LeMay and C. A. Gunter, "Cumulative attestation kernels for embedded systems," in *Proc. 14th Eur. Symp. Res. Comput. Security, ser. ESORICS*, Saint Malo, France, Sep. 2009, pp. 655–670.

[7] Pike Research, "Smart meter installations to reach 250 million worldwide by 2015," Nov. 2009 [Online]. Available: http://www.pikeresearch.com/newsroom/smart-meterinstallations-to-reach-250-million-worldwide-by-2015

[8] M. LeMay, G. Gross, C. Gunter, and S. Garg, "Unified architecture for large-scale attested metering," in *Proc. 40th Annu. Hawaii Int. Conf. Syst. Sci., ser. HICSS '07*, Waikoloa, HI, Jan. 2007.

[9] *Modern Solutions for Protection, Control and Monitoring of Electric Power Systems*, H. J. A. Ferrer and E. O. Schweitzer, Eds. Pullman, WA: Schweitzer Eng. Labs., Inc..

[10] EPRI IntelliGrid Consortium, Automatic Meter Reading (AMR) 2004 [Online]. Available: http://www.intelligrid.info/IntelliGrid_Architecture/Use Cases/CS AMR Use Cases.htm
[11] S. Borenstein, M. Jaske, and A. Rosenfeld, "Dynamic pricing, advanced metering and demand response in electricity markets," UC Berkeley Center for the Study of Energy Markets, University of California Energy Institute, 2002.
[12] "Federal Energy Regulatory Commission Staff Report, Assessment of demand response and advanced metering," 2008.
[13] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, *Maude Manual (Version 2.1)*. Menlo Park, CA: SRI Int., 2005.
[14] R. Anderson and M. Kuhn, "Low cost attacks on tamper resistant devices," in *Proc. 5th Int. Workshop Security Protocols*, Paris, France, Apr. 1997, pp. 125–136.
[15] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Proc. 3rd Workshop Cryptographic Hardware Embedded Syst., ser. CHES '01*, Paris, France, May 2001, pp. 251–261.
[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *Proc. 29th IEEE Symp. Security Privacy, ser. Oakland '08*, Oakland, CA, May 2008, pp. 296–310.
[17] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. 18th USENIX Security Symp., ser. Security '09*, Montreal, QC, Canada, Aug. 2009, pp. 383–398.
[18] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proc. 15th ACM Conf. Comput. Commun. Security, ser. CCS '08*, Alexandria, VA, Oct. 2008, pp. 15–26.
[19] A. Liu and P. Ning, TinyECC: Elliptic Curve Cryptography on TinyOS Feb. 2011 [Online]. Available: http://discovery.csc.ncsu.edu/software/TinyECC/
[20] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul. (TOMACS)*, vol. 8, pp. 3–30, Jan. 1998.
[21] S. A. Vanstone, "Next generation security for wireless: Elliptic curve cryptography," *Comput. Security*, vol. 22, no. 5, pp. 412–415, Jul. 2003.
[22] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The maude LTL model checker," in *Proc. 4th Int. Workshop Rewriting Logic Its Appl., ser. WRLA '02*, Pisa, Italy, Sep. 2002, pp. 162–187.
[23] *Common Language Infrastructure (CLI)*, ECMA-335, ECMA Int., 2006.
[24] M. D. LeMay, "Compact integrity-aware architectures," Ph. D. dissertation, Univ. Illinois at Urbana-Champaign, Urbana, IL, 2011.
[25] M. 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, 2004.
[26] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Security, ser. CCS '07*, Alexandria, VA, Oct. 2007, pp. 552–561.
[27] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Security, ser. CCS '05*, Alexandria, VA, Nov. 2005, pp. 340–353.
[28] Mobile trusted module specification, version 1.0. Trusted Computing Group, Inc., 2010.
[29] J. Winter, "Trusted computing building blocks for embedded Linux-based ARM trustzone platforms," in *Proc. 3rd ACM Workshop Scalable Trusted Comput., ser. STC '08*, Fairfax, VA, Oct. 2008, pp. 21–30.
[30] W. Hu, H. Tan, P. Corke, W. C. Shih, and S. Jha, "Toward trusted wireless sensor networks," *ACM Trans. Sensor Netw.*, vol. 7, pp. 5:1–5:25, Aug. 2010.
[31] D. Zhang and D. Liu, "Dataguard: Dynamic data attestation in wireless sensor networks," in *Proc. 40th IEEE/IFIP Int. Conf. Dependable Syst. Netw., ser. DSN '10*, Chicago, IL, Jun. 2010, pp. 261–270.
[32] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *Proc. 15th Eur. Symp. Res. Comput. Security, ser. ESORICS '10*, Athens, Greece, Sep. 2010, pp. 643–662.
[33] S. McLaughlin, D. Podkuiko, A. Delozier, S. Miadzvezhanka, and P. McDaniel, "Embedded firmware diversity for smart electric meters," in *Proc. 5th USENIX Workshop Hot Topics Security, ser. HotSec '10*, Washington, DC, Aug. 2010, pp. 1–8.
[34] A. Seshadri, A. Perrig, L. v. Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Proc. 25th IEEE Symp. Security Privacy, ser. Oakland '04*, Oakland, CA, May 2004, pp. 272–282.
[35] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. 16th ACM Conf. Comput. Commun. Security, ser. CCS '09*, Chicago, IL, Nov. 2009, pp. 400–409.
[36] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Enabling intrusion analysis through virtual-machine logging and replay," in *Proc. 5th Symp. Oper. Syst. Design Implementation, ser. OSDI '02*, Boston, MA, Dec. 2002, vol. 36, pp. 211–224.
[37] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh, "Dejaview: A personal virtual computer recorder," in *Proc. 21st ACM Symp. Oper. Syst. Principles, ser. SOSP '07*, Stevenson, WA, Oct. 2007, pp. 279–292.
[38] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *Proc. 21st ACM Symp. Oper. Syst. Principles, ser. SOSP '07*, Stevenson, WA, Oct. 2007, pp. 189–204.
[39] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems," in *Proc. 6th USENIX Symp. Netw. Sys. Design Implementation, ser. NSDI '09*, Boston, MA, Apr. 2009, pp. 1–14.
[40] A. Seshadri, M. Luk, A. Perrig, L. v. Doorn, and P. Khosla, "SCUBA: Secure code update by attestation in sensor networks," in *Proc. 5th ACM Workshop Wireless Security, ser. WiSe '06*, Los Angeles, CA, Sep. 2006, pp. 85–94.
[41] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. v. Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proc. 20th ACM Symp. Oper. Syst. Principles, ser. SOSP '05*, Brighton, U.K., Oct. 2005, pp. 1–16.
[42] P. E. Lanigan, R. Gandhi, and P. Narasimhan, "Sluice: Secure dissemination of code updates in sensor networks," in *Proc. 26th IEEE Int. Conf. Distrib. Comput. Syst., ser. ICDCS '06*, Lisboa, Portugal, Jul. 2006.
[43] J. Herzog, "Applying protocol analysis to security device interfaces," *IEEE Security Privacy*, vol. 4, pp. 84–87, Jul. 2006.
[44] C. Bernardeschi, A. Fantechi, and S. Gnesi, "Model checking fault tolerant systems," *Softw. Test., Verification, Rel.*, vol. 12, no. 4, pp. 251–275, Dec. 2002.
[45] G. Klein, "Operating system verification-an overview," *Sadhana*, vol. 34, pp. 27–69, 2009.
[46] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proc. 22nd ACM Symp. Oper. Syst. Principles, ser. SOSP '09*, Big Sky, MT, Oct. 2009, pp. 207–220.

Michael LeMay received the B.S. degree in computer science from the University of Wisconsin-Eau Claire in 2005 and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 2011.

He is currently a Postdoctoral Research Associate with the Information Trust Institute, University of Illinois at Urbana-Champaign. His research interests are in trustworthy computing and embedded system security, particularly for smart grid devices and healthcare IT.

Carl A. Gunter (M'94–SM'95) received the B.A. degree from the University of Chicago, IL, in 1979 and the Ph.D. degree from the University of Wisconsin, Madison, in 1985.

He is currently a Professor at the University of Illinois at Urbana-Champaign, where he is the director of the Illinois Security Lab. His research interests are in security, networks, programming languages, and software engineering.

Dr. Gunter is a Senior Member of the IEEE Computer Society.