Securing Dynamic Firmware Updates of Mixed-Critical Applications

George Kornaros and Svoronos Leivadaros
Informatics Engineering Department
University of Applied Sciences Crete
71004 Heraklion, Crete, Greece
Email: kornaros@ie.teicrete.gr

***Abstract*—This work introduces a secure framework for run-time updating of firmware in Internet of Things devices that execute mixed-critical applications. Taking advantage of the capabilities of modern heterogeneous System-on-Chip devices to run cores in asymmetric multiprocessing (AMP) configuration, we developed a methodology to showcase dynamic updating of real-time applications in a novel secure way when executing on a Xilinx ZYNQ-based platform. As an exemplary implementation we demonstrate a bio-signal monitoring use case that reads accelerometer data to determine if a person has fallen, while a distant medical management system can dynamically perform firmware updates. Even sophisticated code injection or reuse attacks can be subverted with the proposed defenses that ensure a practically isolated environment for the critical firmware with negligible overhead to the device in terms of performance and cost.**

Keywords– Secure firmware update, Asymmetric embedded multriprocessing, trusted dynamic realtime update

1. Introduction

With the advent of new technologies to develop smart devices and wide deployment of WI-Fi networks, Internet of Things (IoT) is growing at a very fast pace. IoT devices can be any kind of computing system but is usually a type of embedded system such as smartphones, household appliances, sensors and actuators to monitor environmental conditions and act accordingly, surveillance cameras with pattern recognition capabilities, weather monitoring systems and many others.

However, security issues have arisen regarding the protection of sensitive data handled by some of the devices connected to the Internet of Things, the devices themselves or the gateways that connect IoT devices to company and manufacturer networks. Until a few years ago, security wasn't a high priority since manufacturers of embedded devices utilizing the IoT relied on the concept of *security by obscurity* to avoid malicious third-parties from hacking into and stealing sensitive data or tampering with devices that execute critical applications. IoT security concerns various aspects, such as securing of on-device confidential or private data, security of connected devices' communication links or their interfaces themselves, authenticity of software and updates and protection of intellectual property.

Security breaches have been reported in the recent past exploiting system vulnerabilities. Between 2010 and 2014, Cyber attackers successfully compromised the security of U.S. Department of Energy computer systems more than 150 times [1]. Miller and Valasek showcased a weakness in the CAN bus connecting the Electronic Control Units of the Fiat Chrysler Automobile by remotely connecting to the car's system attacking communications [2]. Glissom et al., investigated the viability of compromising a mannequin patient used in medical simulation training environment [3]. As shown by the results of this work, only moderate knowledge in computer science and information technology is needed to successfully compromise the security of a medical device with connectivity capabilities. The connectivity of these devices to a public network can compromise the security of the device if no countermeasures are employed such as using authentication mechanisms and data encryption.

In this scope we developed a framework to resolve end-to-end security issues that the IoT interconnected devices face while supporting run-time firmware updates. Today, modern Systems-on-Chip (SoCs) typically employ heterogeneous compute resources, or even processor components in asymmetric multiprocessing (AMP) configurations, which are called *remote* processors. This strategy inherently enables strong isolation of compute resources. These remote AMP processors are commonly used to control latency-sensitive sensors, or drive random hardware blocks in support of real-time behavior [4][5]. Additionally, AMP mode can also offer a significant advantage of a securely executing application by significantly reducing the ability of adversaries to tamper this code in this isolated environment. However, since any firmware running on the remote processor has access to the whole of system memory and is in kernel mode, it effectively runs with Linux root privileges [6]. Thus, any interaction with firmware should be subject to the necessary security policy in the target system to prevent exploits.

In this work we propose architecture extensions in AMP organizations that rise up to the device stack to continuously maintain the trusted computing base. The main contribution involves a method and infrastructure to support trustworthy dynamic updating of key parameters and firmware of time-critical applications in a mixed-critical environment, through using standard encryption and authentication control mechanisms with negligible overheads to the device in terms of performance and cost. While in this paper we focus on Internet of things devices and on evaluating our method using a health-care use case, there are many additional

application areas, e.g. in cloud computing.

2. Related Work

Many academic researchers and information technology companies have released studies on security issues of IoT. Sachin Babar et al. [7]analyze the issue of security in IoT. They divide attacks/security breaches that an IoT device may receive, into categories. They go through the necessary security principles that a system needs to implement to ensure secure operation and data protection. Rolf H. Weber [8] covers the new security challenges that the emergence of IoT has brought with it. He outlines the requirements that a IoT-based system must meet and describes the legal and legislator aspect of the IoT.

Mentor Graphics has released the Mentor Embedded Multicore Framework (MEMF) [9], a commercial multicore system development framework based on the OpenAMP that allows embedded system developers to deploy multiple operating systems and applications across homogeneous or heterogeneous multicore processors. Xilinx has released works [26][29] describing principles and configurations that developers should adopt on their products to ensure secure booting and operation of an embedded system, but with no uptake. While hardware-supported Trusted Execution Environments (TEE) using TI's M-shield [10] and ARM Trustzone [11] are widely deployed today, they are typically constrained in terms of code and data memory. For instance SSL/TLS control is hardly performed within TEE due to resource constraints.

Intel's TXT extensions [12] provide a measured and controlled launch of system software that will then establish a protected environment for itself and any additional software that it may execute. Similarly, Trusted Computing Group (TCG) [13] defines a Root of Trust for Measurement (RTM) that executes on each platform reset; Further, TCG defined Dynamic Root of Trust Measurement in Trusted Computing systems in its specification as a technology for measured platform initialization while system is in running state. Flicker [14] and Trust Visor [15] are solutions that implement DRTM. In synergy with these technologies we propose hardware assisted methods that also remove the slow Trusted Platform Module hardware from the systems' critical path.

Researchers have also proposed hardware memory protection schemes for low-cost devices [16]. These can be programmed in software, while they allow flexible management of memory and peripheral I/O regions without burdening the CPU. However, Trustlite requires all software components to be loaded and configured at boot time, while our proposal supports dynamic secure updates.

To secure embedded devices against software vulnerabilities various works offer efficient solutions mostly in OS level, such as BINtegrity [17], which addresses mitigation of memory corruption attacks. However, it requires kernel-based runtime support resulting in OS overheads, and hence it is hardly useful in real-time critical applications. C-FLAT [18] is a mechanism that allows precise verification of an application's control flow to detect attacks on embedded systems that are increasingly deployed in safety-critical infrastructures. To achieve application isolation, virtualization is widely adopted where the security critical point is the hypervisor and its sensitivity to attacks [19][20]. Thus, the hypervisor must be part of the Trusted Computing Base (TCB), with reduced interactions and commonly require significant hardware support to perform efficient resource distribution and isolation. Complementary to such solutions, but with reduced requirements in costs, we support isolated environment and authentication when updating a critical application or when distant communication with a critical application is conducted. Our contributions focus in avoiding system calls with non-encrypted information among "normal" and "secure" worlds (e.g. in ARM-based TEE) and in avoiding resulting derived keys to live in normal user-space RAM.

3. Preliminaries in Security Concerns

To tackle the aforementioned security issues we developed a framework to process and isolate sensitive data on an embedded IoT device and most importantly to support dynamic firmware updates in a secure and trustworthy way. In general, the input and output data during the operation of a system can be categorized in three types of data that need to be secured:

- Data sent remotely from a distant device to our host device (Transmitted Data)
- Data transferred using memory-to-processor and processor-to-processor links (Processed Data)
- Data stored in storage medias used for the operation of a device (Stored Data)

The reason we classified our data is because different types of data need different methodologies to secure. Transmitted data need to be encrypted before sent to a device in order to prevent man-in-the-middle attacks. To this end in our framework the connection between the devices uses SSH-based authentication and authorization to send out updates and new thresholds from the distant management system to any bio-metric device. Initially, data to send are encrypted by using the AES encryption algorithm with a key available only to the distant management system and to the bio-metric device. Finally, the received message to the device is decrypted and also authenticated by using an HMAC algorithm.

Processed data (with potential associated properties of deadlines in their processing or authenticity/integrity) need to be isolated from the data handled by a conventional OS. For this reason we used the OpenAMP framework [11][12] to develop a system running in Asymmetric Multi-Processing mode to allow one processor to run a Linux system and another processor to run Real-Time applications handling sensitive data.

Stored data should be stored encrypted and only decrypted when we need them in our applications. Sensitive data in our system cannot be accessed by non-legitimate users.

3.1. Threat Model

We consider an infrastructure of networked nodes, where each node consists of a low-end microcontroller. These nodes can be subject of attacks that aim at executing unauthorized privileged code. The attack can include injection of malicious code or escalation of the privilege level of user space binaries. Since our nodes support dynamic updating of parameters of the critical soft (or even hard) real-time firmware, or updating of the firmware itself, we also consider man-in-the-middle attacks and adversaries that attempt to impersonate a trusted distant infrastructure manager. Our system can prevent the damage that can result from having a compromised OS, a hijacked communication link, or unauthorized firmware credentials. Encrypted data, encryption keys that are typically stored in volatile memory (DRAM) are also vulnerable to unauthorized hacking, warm or cold-boot attacks [21]. Our proposed method enables dynamic firmware updates in a secure way, without the need to design tamper detection and scrubbing circuits for protection against such type of attacks. Even if the adversary has full knowledge of the application's memory layout and has full control over the program's stack and heap to inject new return addresses (i.e., return oriented programming [30]), we show that in our framework the firmware can still run in a secure isolated memory space. In this work we consider cache-based timing attacks out the scope, or side-channel attacks such as differential power analysis or differential electromagnetic analysis.

3.2. Use Case

We evaluate our framework through implementing a Biosignal Monitoring and Remote Medical Management System, which allows us to run software on devices equipped with tri-axial accelerometer sensors that can detect falls. In addition, this system implements a distant management and updating system that allows an authorized individual such as a doctor to update critical parameters of the bio-metric device, e.g. the accelerometer thresholds which determine the device's fall detection sensitivity (and alert indications triggered by a fall).

The algorithm used to detect falls is described by Y.He et al. [22]. This algorithm extracts three key values from tri-axial accelerometer data and compares these values with specific thresholds in order to detect activity levels of individuals and whether someone has fallen or not. The three key values are:

- Signal Magnitude Area (SMA): describes the degree of change of human motion at any given time (sample). High values indicate sudden change in motion
- Signal Magnitude Vector (SMV): describes the vector sum of acceleration a person feels in at any given time (sample). High values indicate high total acceleration
- Tilt Angle (TA): describes the angle of the person in degrees at any given time (sample)

The application reads samples in discrete time intervals. The discrete equations that compute these values for the i-th sample of a data set follow.

$$SMA_i = \frac{1}{i}(\sum_{u=1}^{i}|x_u| + \sum_{u=1}^{i}|y_u| + \sum_{u=1}^{i}|z_u|) \quad (1)$$

$$SMV_i = \sqrt{{x_i}^2 + {y_i}^2 + {z_i}^2} \quad (2)$$

$$TA_i = arcsin(\frac{y_i}{SMV_i}) \quad (3)$$

In order to detect a fall, we compare each sample from our data with a specific threshold value. If, for a given sample, all three key values exceed a certain threshold, it is assumed that the person wearing that device has fallen.

In our application, the three types of data in regard to the classification are:

- New thresholds for the fall detection algorithm (Transmitted Data)
- Accelerometer data being read and processed to deduce whether a person has fallen or not (Processed Data)
- The accelerometer data and threshold values, stored in files for emulation or bookkeeping reasons (Stored Data)

4. System Architecture and Implementation

We developed the fall-detection application in a Zynq-based Zedboard platform. The Zynq Z7020 SoC on the Zedboard contains a dual-core 667MHz ARMv7 32b Cortex-A9 CPU with a NEON SIMD unit along with a 53.2K LUT, 220 DSP-block, 560 kB BlockRAM FPGA connected over AXI port rated at 800 MB/s. The Zynq processing system runs Petalinux [28] using a version 4.0.0 Linux kernel in an Asymmetric MultiProcessing (AMP) configuration.

In an AMP configuration, the processors are assigned their own memory regions, which allows data isolation. An inter-process communication (IPC) protocol is important for exchanging info between the different processors of the AMP configuration as shown in figure 1. The device running in AMP mode uses a Master-Slave communication model, in the sense that one processor or cluster of processors (the master) is responsible for managing the life cycle of the other processor or cluster of processors (the slave/s).

The OpenAMP framework utilizes two key components to establish an AMP configuration, the Remoteproc framework[23] and the RPMsg framework [24]. The remoteproc framework allows a master processor to control (power on, load firmware, power off) the remote (slave) processors while abstracting the hardware differences. The RPMsg protocol is a virtio-based shared memory protocol for inter-process communication. It uses a shared buffer to allow exchange of data between two processors. Currently the maximum size of the buffer's size is limited to 512 bytes. We implemented the AMP configuration of the system by using the OpenAMP framework [25][27] to run

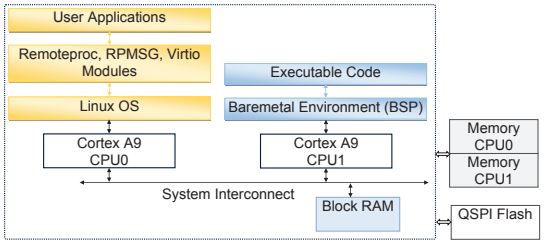

Figure 1. Architecture layout of the Zedboard-based system running in AMP mode; off-chip QSPI Flash stores the first and second stage boot loaders along with the kernel.

Linux on one processor of the Zedboard for non-critical applications and the bio-metric processes needed to detect a fall on the other processor. Petalinux was configured to operate the Zedboard in AMP configuration while also enabling RPMsg and remoteproc modules, and we included the Dropbear/SSH package to enable secure file transmission between Remote Management System and the Zedboard platform.

4.1. AMP Linux/Baremetal System Architecture

CPU0 loads Linux from the rootfs partition of the SD card and is responsible for booting CPU1 and loading the remote monitoring application and handling basic I/O operations that allow CPU1 to read a file. CPU1 is responsible for reading the accelerometer data, for running the fall detection algorithm and printing the data on the console output. CPU1 and CPU0 have access to different parts of the system memory. The segregated allocation of memory to our processors allows us to deny access of one processor to the memory assigned to the other, thus isolating and protecting the sensitive data and software running on CPU1 from the non-critical applications running on Linux on CPU0.

Both processors use a small amount of shared memory defined by the RPMsg protocol, a virtio-based inter-process communication protocol that the OpenAMP framework uses to facilitate data transmission between CPU0 and CPU1 in kernel space. This shared memory is used to perform firmware updates transferring the thresholds from the master monitor application part running on CPU0 to the remote monitor application running on CPU1. The accelerometer data are typically read directly by the CPU1 from the accelerometer peripheral sensors. However, we emulated this process by feeding data to the CPU1 through the RPMsg channel.

CPU0 uses the *remoteproc* protocol to boot CPU1 and load the remote monitoring application. After CPU0 initializes CPU1, CPU1 runs the remote monitoring application on the memory and returns a name service back to CPU0. Next, the master application allocates the RPMsg buffer for shared memory and uses the name service as an identifier to link the two processes. Basic I/O operations executed by the remote processor will be carried out by the master processor and their data will be written on the RPMsg device and read by CPU1.

In practice, the device is activated, i.e., the fall detection process begins when the middle button on the Zedboard is pressed. Initially, the three threshold key values are retrieved by requesting the encrypted *thresholds.txt* file stored in the filesystem. CPU1 then copies the ciphertext to its allocated memory and then decrypts the ciphertext using the same private key used when the data was encrypted on the distant management system. The application reads pre-sampled tri-axial accelerometer data which are stored in text files in the following format:

```
-0.795;9.87;0.393
-0.804;9.84;0.373
-0.804;9.8...
```

where each line is one sample and the ';' symbol is used to split each sample to three separate values. Each value is the acceleration measurement in the X, Y, Z axis respectively.

These custom values, which are actually accelerometer data captured in a file, are stored in the root home directory of the Linux system running on CPU0 of the Zedboard. The remote application requests the accelerometer data located in the filesystem of CPU0. After reading the data, the remote application enters a loop, where a monitoring process is activated by calling the fall_detect() function. The result is printed in the console through the RPMsg buffer. We can dynamically update the algorithm thresholds using the distant management service that we developed (as shown in Fig. 2). The new thresholds take effect after decryption and successful authentication.

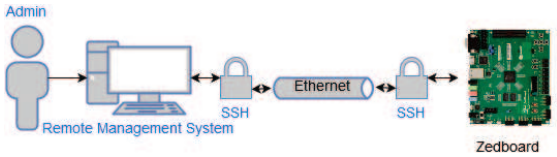

Figure 2. Distant medical management system layout

4.2. Medical Management System Architecture

The distant management system can be any Linux-based operating system with the SSH package installed. For this implementation we used the development host machine as the distant medical manager. Because we included the Dropbear/SSH package on the Zedboard GNU/Linux OS when we configured our Petalinux project, we are able to establish a secure shell connection to remotely manage the device.

We developed an automated way (via scripting commands) that allows an authorized user to input new SMA, SMV and TA thresholds. The new thresholds are then parsed to a thresholds.txt file, which is signed by the OEM by adding an HMAC digest. It is then encrypted using the AES algorithm, and sent to the device through using the SSH protocol. This is the file that the remote application on CPU1 opens when it needs the thresholds for the fall detection algorithm. Notice, that authentication services may be supported by the medical manager and not by the manufacturer.

To use the thresholds, CPU1 uses the same private key, which was used to encrypt the thresholds file on the server, to decrypt the file. Then, CPU1 uses the plaintext created to generate an HMAC digest which is then compared to the original generated digest in order to detect whether someone tampered with the data during transmission.

The thresholds update functionality can run concurrently with the fall detection application. Thus, no system reset is needed since the new thresholds take effect immediately.

5. System Operation

When the development board boots from the external QSPI Flash (figure 1), u-boot initializes the AES encryption key and the HMAC key on an internal BRAM inside the programmable logic. While secure booting is supported in Xilinx Zynq-based platforms[1], the zedboard does not support encrypted bitstream programming. Only CPU1 has immediate access to this BRAM and uses these private keys when a critical parameters update is applied. The same keys are used on the distant management system (provided by the manufacturer) to ensure the secure update of the thresholds.

We run the fall detection algorithm using the middle push button on the Zedboard to initiate the process. The console output of the device reports as follows in Figure 3.

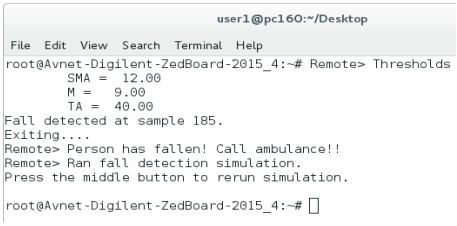

Figure 3. Output report of fall-detection application

5.1. Distant Management System Operation

After verifying the fall detection application functions correctly while providing a predictable timing behavior (since it is executed in CPU1), we executed the Remote Management Service on the Distant Development Machine (Fig. 2) to dynamically alter the thresholds. Figure 4 depicts the console output of the thresholds update service.

Figure 5 shows the flow diagram that describes the steps required to perform the transmission of the new thresholds in order to ensure the confidentiality and integrity of the data. The distant management system transmits the update packet to the device manufacturer to generate the HMAC, which in turn is encrypted in order to remotely perform the firmware update. The device contains the secret key originally set by the manufacturer which is needed to authenticate the firmware updates. Hence, when the new update packet is received initially it is decrypted using the secret key that is shared between the distant manager and the device and then the decrypted data are authenticated.

1. through using the battery-backed RAM or the eFUSE array (an on chip one time programmable (OTP) non-volatile memory)

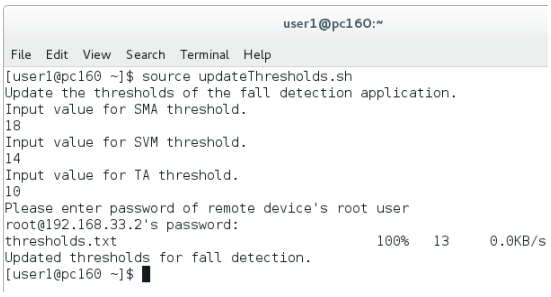

Figure 4. Output report of threshold update service

An incoming update operation cannot be tampered since the encryption keys are only accessible by the remote processor (CPU1); essentially, CPU0's memory map does not include the secret storage in the programmable logic. Essentially, the Operating System on top of CPU0 has no insight whatsoever of any resource located inside the programming logic. If a man-in-the-middle interferes, or a violation is attempted by an unauthorized-untrusted application that runs on CPU0, then the update packet that is delivered to CPU1 is just discarded.

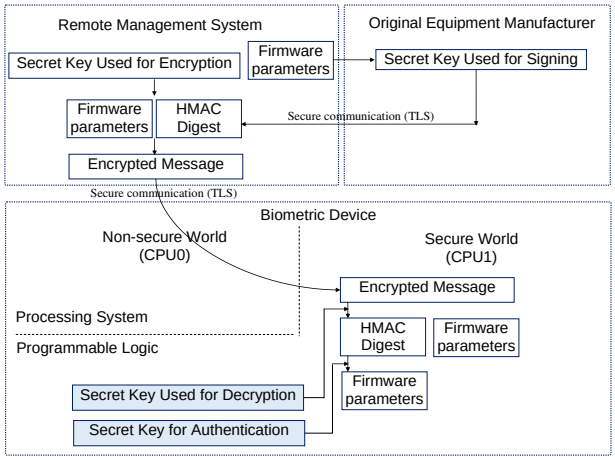

Figure 5. Flow diagram of the threshold update procedure

After multiple attempts to send unauthorized packet updates to CPU1 we rerun the fall detection application on the Zedboard without resetting the system, but only a single trusted update is validated and the algorithm thresholds changed. Figure 6 depicts the validation result. The thresholds take effect immediately after authentication, causing the fall detection algorithm to avoid detecting a fall.

The decryption and authentication algorithm of the thresholds costs 410 μsec for the Cortex-A9, while the fall-detection algorithm requires more than 36 msec to provide a decision. To accommodate for other use-cases the authentication process should not be a burden to the time constraints of the time critical application. Thus, we integrated a SHA-256 hardware engine, accessible only by the CPU1. The SHA-256 accelerator implements the standard SHA algorithm, defined in the national institute of standards

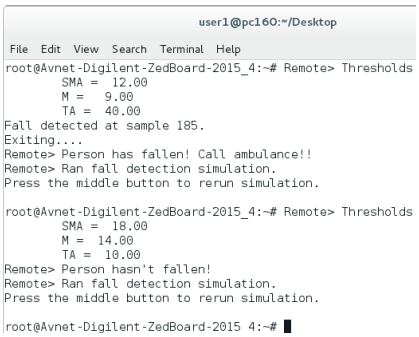

Figure 6. Output of Fall Detection Algorithm after updating the thresholds

and technology (NIST) as a U.S. federal information processing standard (FIPS) 180-3. The SHA-256 engine digest a message down to a 256-bit result in less than 70 10ns clock cycles.

5.2. Secure Firmware Update

To achieve hardware isolation and attack-shielded dynamic updates, we developed the organization shown in figure 7. The key feature is that both the keys and the critical code itself are never present in the platform RAM that is directly accessible by the "normal" world or by external attacks to system memory. Hence, this extension enables a critical application to defend itself from a tampered OS.

The encrypted message is streamed to the hardware decrypt and it is buffered locally to further examine the computed SHA-256 digest. Two block RAMs (BRAMs) are maintained inside the programmable logic to store the trusted binary code; these BRAMs are memory-mapped only to CPU1's address space. Essentially, one BRAM buffer serves as a shadow memory to store a new firmware code while it is authenticated; at the same time CPU1 executes the currently active firmware that is located in the sibling buffer. The two buffers are used interchangeably in a seamless way, since the CPU1 is notified by a returned address pointer in the active buffer.

After CPU1 delivers the encrypted message to the hardware secure engine, CPU1 examines the status register of the hardware secure engine to identify when the new update is ready, if authentication is successful. When it is validated, the function base address is retrieved and CPU1 invokes the new code from the internal BRAM. Even if an evildoer manages to gain control of CPU1, there is no path for CPU1 to write to its internal code memory, the BRAM buffer. The hardware write port of this BRAM buffer is internally controlled by the secure engine. Hence, notice that *only after authentication a new firmware can be transferred in the BRAM*. Our low-level driver of the hardware secure engine provides the next application programming interface call to the CPU1.

```
status_t compute\_SMA (
  sma_value_t (*callback)(accelerometer_t agent,
                          void *data), void *data )
```

As a proof of concept we developed the critical computation of the SMA value (i.e., equation (1)) in two different codes. When the device boots the first version of the SMA code runs and during the operation we transmit an update with the second version of the SMA function. CPU1 after validating the new update in 2.16ms, CPU1 discovers the trusted/secure function to compute the Signal Magnitude Area (SMA) value in the internal BRAM and then uses this callback function pointer to process the current data samples.

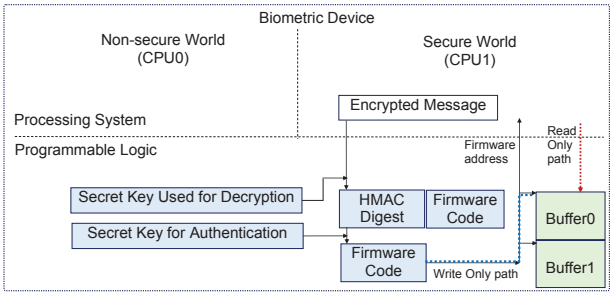

Figure 7. Hardware supported firmware updating through full decryption and authentication and code RAM configuration in hardware; notice that CPU1 has only read access to code RAM by hardware construction, while intermediate code variables of equations (1),(2) and (3) are kept in registers.

Additionally, inspired by the Trusted Platform Module (TPM) specification of the Trusted Computing Group (TCG) [13] we developed a Device Configuration Register, hereby named DCR, to securely store and attest the state of the device by essentially maintaining a device firmware update metric. The particular integrity metric that forms the device state is measured as cryptographic hash. Actually the DCR can not be simply overwritten with a new value, but only be extended with a new measurement. The new value of the extended DCR is computed as follows: DCR_{NEW} = HF(DCR_{OLD} $||$ value to add) where HF is a cryptographically secure hash function with a block size of 256-bits. In this alternative realization it is computed by the SHA-256 hardware core. Through this metric the distant manager can identify the device status. The hardware cost is dominated by the area occupied by the SHA-256 core of 2412 LUTs.

Despite the successful progress of modern techniques such as return-oriented programming (ROP) [30] that can easily circumvent executable space protection in both x86 (NX bit) and ARMv8-M architecture (eXecute-Only-Memory, XOM [31]), our proposed method ensures the firmware to run in isolated private memory space and only after it has been cryptographically validated. Our target to achieve runtime firmware updates is free from code reuse attacks and thus free from complex software or hardware solutions against ROP like control-flow integrity (CFI) to restrict program execution to a pre-defined control-flow graph (CFG) or code randomization (e.g., ASLR). These can still have impact to the Linux environment but there is no way to bypass the proposed defense mechanism on the isolated firmware.

6. Conclusions

There are three main methods by which our data are secured in this implementation. First, by using the OpenAMP framework we segregated the memory available between the two processors. This makes it harder for malware running on the memory space of CPU0 to corrupt data and programs handled in the memory space of CPU1, providing a way to protect our processed data. Second, we use the AES algorithm to encrypt the new thresholds and firmware on the distant management system before we send it through using secure communication. The update is delivered in encrypted form to the remote processor (CPU1) and decrypted only when needed by CPU1, with the assistance of custom crypto circuitry that works in parallel while the real-time code runs. In order to ensure data integrity and authenticity, we used the HMAC algorithm before integrating either new thresholds or new firmware. The private keys are loaded on the development board's BRAM during the U-Boot loading stage. Hardware accelerated integrity and authentication are controlled by the sealed environment of CPU1, thus providing a minimized trusted code base (TCB) and isolation from the legacy OS. Additional layers of security can be added to our system, to enhance providing a dynamic root of trust for measurement (DRTM), so that the system can be attested for integrity at run-time. We anticipate the support of multiple protection containers and implementing mechanisms for authenticating a connection for machine-to-machine communication; we consider encrypting transmitted data with different light-weight encryption algorithms taking into account the capabilities that IoT devices have in computing power or the requirements in low power consumption, or low availability of computing resources.

Acknowledgment

This research was supported by funding from the European Union (EU) Horizon 2020 project TAPPS (Trusted Apps for open CPSs) under RIA grant No 645119.

References

[1] S. Reilly, 2015, USA Today, (2015, Sep 11) Records: Energy Department struck by cyber attacks http://www.usatoday.com/story/news/2015/09/09/cyber-attacks-doe-energy/71929786/.

[2] A. Greenberg, 2015, (2015, July 21) Hackers Remotely Kill a Jeep on the Highway With Me in It https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway.

[3] W. B. Glisson, J. T. McDonald, T. R. Andel, S. M. Campbell, M. Jacobs, and J. Mayr, "Compromising a medical mannequin", in *P. Rico: Americas Conference on Information Systems (AMCIS)*, 2015.

[4] M. Redfearn, "MIPS remote processor driver for managing linux and real-time processing," in *Embedded World 2017 Conference*, 2017.

[5] Samsung Semiconductor Inc., 2016, Samsung Bio-Processor For Health Monitoring White Paper.

[6] Imagination Technologies Limited, 2016, MIPS OS Remote Processor Driver Whitepaper.

[7] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for internet of things (IoT)," in *2nd International Conference Wireless VITAE*, pp. 1–5, 2011.

[8] Weber R. H., "Internet of things - new security and privacy challenges," *Comput. Law Secur. Rev.*, vol. 26, no. 1, pp. 23–30, 2010.

[9] Mentor, 2016, mentor Embedded Multicore Framework (MEMF), https://www.mentor.com/embedded-software/multicore.

[10] J. Azema and G. Fayad, 2008, m-Shield mobile security technology: Making wireless secure. Texas Instruments, White Paper.

[11] ARM, 2009, ARM Security Technology, Building a Secure System using TrustZone Technology, White Paper PRD29-GENC-009492C.

[12] J. Greene, 2016, Intel Trusted Execution Technology, WhitePaper, www.intel.com/txt.

[13] Trusted Computing Group, 2014, Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.16 Oct. 2014.

[14] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys'08, 2008, pp. 315–328.

[15] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, 2010, pp. 143–158.

[16] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, 2014, pp. 10:1–10:14.

[17] M. Neugschwandtner, C. Mulliner, W. Robertson, and E. Kirda, "Runtime integrity checking for exploit mitigation on lightweight embedded devices," in *International Conference on Trust and Trustworthy Computing*, 2016, pp. 60–81.

[18] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, 2016, pp. 743–754.

[19] R. J. Masti, C. Marforio, K. Kostiainen, C. Soriente, and S. Capkun, "Logical partitions on many-core platforms," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC'15, 2015, pp. 451–460.

[20] D. Rosenberg, 2014, QSEE TrustZone kernel integer over flow vulnerability, In Black Hat Conf.

[21] J. A. Halderman, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, May 2009.

[22] Y. He, Y. Li, and C. Yin, "Falling-incident detection and alarm by smartphone with multimedia messaging service (mms)," *E-Health Telecommunication Systems and Networks*, vol. 1, no. 1, 2012.

[23] The Linux Kernel Organization Inc., 2016, Remoteproc Framework Linux Documentation, www.kernel.org/doc/Documentation/remoteproc.txt.

[24] The Linux Kernel Organization Inc., 2016, RPMsg framework Linux Documentation, www.kernel.org/doc/Documentation/rpmsg.txt.

[25] MCA, 2016, multicore Association Working Groups, http://www.multicore-association.org/workgroup/oamp.php.

[26] Xilinx, 2015, Leveraging Asymmetric Authentication to Enhance Safety-Critical Applications, xilinx.eetrend.com/files-eetrend-xilinx/download/201512/9574-21502-wp468asym-auth-zynq-7000.pdf.

[27] Xilinx, 2016, Xilinx wiki, OpenAMP www.wiki.xilinx.com/OpenAMP.

[28] Xilinx, 2016, Petalinux SDK Design Tools, www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[29] Xilinx, 2015, Secure boot of Zynq-7000 All-Programmable SoC http://japan.zylinks.com/support/documentation/application notes/xapp1175 zynq secure boot.pdf.

[30] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS'10, 2010, pp. 559–572.

[31] J. Yiu, 2015, ARMv8-M Architecture Technical Overview Whitepaper.