University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

March 2020

TRUSTWORTHY SYSTEMS AND PROTOCOLS FOR THE INTERNET OF THINGS

Arman Pouraghily
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Computer and Systems Architecture Commons, Hardware Systems Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation

Pouraghily, Arman, "TRUSTWORTHY SYSTEMS AND PROTOCOLS FOR THE INTERNET OF THINGS" (2020). *Doctoral Dissertations*. 1864.
https://scholarworks.umass.edu/dissertations_2/1864

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

TRUSTWORTHY SYSTEMS AND PROTOCOLS FOR THE INTERNET OF THINGS

A Dissertation Presented

by

ARMAN POURAGHILY

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2020

Department of Electrical and Computer Engineering

© Copyright by Arman Pouraghily 2020

All Rights Reserved

TRUSTWORTHY SYSTEMS AND PROTOCOLS FOR THE INTERNET OF THINGS

A Dissertation Presented

by

ARMAN POURAGHILY

Approved as to style and content by:

Tilman Wolf, Chair

Wayne Burleson, Member

Brian Levine, Member

Russell Tessier, Member

Christopher Hollot, Department Head
Department of Electrical and Computer Engineering

DEDICATION

To my mother and my father, Afsane and Akbar.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest appreciation to my advisor Prof. Tilman Wolf. Thank you for your support and mentorship throughout my entire journey as a Ph.D. student. I learned a lot from you both actively following your advice and passively watching you and learning from you as a role model in my personal and professional life.

I have to also thank professor Russell Tessier with whom I had the pleasure of working on multiple research projects. His knowledge and expertise in the subject matter, as well as his support and guidance all along the path of doing research, were always helpful, especially during the time of facing new challenges in our projects.

This work, without any doubts, would never be possible without the help and contributions of my colleagues and good friends at the Department of ECE. Firstly, I would like to thank Georgios Provelengios, one of the most knowledgable and most humble people I have ever had the opportunity of working with. We had a very successful collaboration only on one project but his knowledge and expertise were always helping me while working on this dissertation. I should also thank Tedy Thomas and Kekai Hu for their contributions and help in the earlier stages of this work.

I also had the pleasure of having wonderful teachers during my Ph.D. from whom I learned a lot. Without them, I would never be the person I am today and obviously, their influence is apparent in my work. It is a pleasure for me to thank Professors Maciej Ciesielski, Csaba Andras Moritz, Wayne Burleson, David Irwin, Michael Zink, and Brian Levine.

I also wish to thank Altera Corporation for the donation of the Quartus II software and DE4 board.

The work presented in this dissertation was also supported by National Science Foundation under Grant Numbers 1115999, 1617458, and 1551444.

ABSTRACT

TRUSTWORTHY SYSTEMS AND PROTOCOLS FOR THE INTERNET OF THINGS

FEBRUARY 2020

ARMAN POURAGHILY

B.Sc., UNIVERSITY OF TEHRAN

M.Sc., UNIVERSITY OF TEHRAN

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Tilman Wolf

Processor-based embedded systems are integrated into many aspects of everyday life such as industrial control, automotive systems, healthcare, the Internet of Things, etc. As Moore's law progresses, these embedded systems have moved from simple microcontrollers to full-scale embedded computing systems with multiple processor cores and operating systems support. At the same time, the security of these devices has also become a key concern.

Our main focus in this work is the security and privacy of the embedded systems used in IoT systems. In the first part of this work, we take a look at the security of embedded systems from a hardware point of view. We describe why we believe current security approaches fall short when it comes to securing modern embedded processors. We propose our hardware monitoring solution and expand it to cover a variety of embedded systems with different architectural specifications and applications.

In the second part, we shift our focus from hardware to software and protocols involved in securing IoT systems and maintaining the privacy of the data they exchange. We argue why conventional financial mechanisms cannot be applied to this context when trying to monetize data sharing. We propose a financial mechanism based on blockchain technology and demonstrate how it can replace conventional methods. We discuss how the high processing demand of such protocols hinders widespread adoption on different IoT systems, mostly ones based on low-end embedded processors. To eliminate that barrier, we propose a novel, lightweight payment verification protocol that uses a hybrid IoT ecosystem based on low-end and mid-range embedded systems that can be horizontally integrated with other ecosystems and exchange data and assets with monetary values such as cryptocurrencies.

The last part of this work is the further expansion of the aforementioned hardware monitoring approach to enable it to secure high-end embedded systems. Using this new hardware monitoring system, we build a prototype IoT system that runs our proposed lightweight payment verification protocol to exchange data and money. By evaluating this system, we illustrate how our hardware and software approaches can be complementary to each other to safeguard IoT devices against remote attacks.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xiii

LIST OF FIGURES xiv

CHAPTER

1. INTRODUCTION 1

1.1 Hardware Monitoring In Embedded Systems 3
1.2 Horizontally Integrated IoT 5
1.3 Organization and Contribution 7

2. WORKLOAD MONITORING IN EMBEDDED SYSTEMS 9

2.1 Related Work 11
2.2 System and Security Model 13

2.2.1 Secure Processing with Hardware Monitors 13
2.2.2 Security Model 14

2.2.2.1 Security Requirements 15
2.2.2.2 Attacker Capabilities 15

2.3 Monitor Design 16

2.3.1 Task Management in the Operating System 16
2.3.2 Multi-Task Hardware Monitor System 17
2.3.3 OS-to-Monitor Interface for Context Switch 19
2.3.4 OS-to-Monitor Interface for Process Creation 20

2.4 Prototype Implementation 22

2.4.1 System Setup 22
2.4.2 Monitor Context Management 23
2.4.3 Attack Detection and Protection 25
2.4.4 Monitoring System Resources 26
2.4.5 Discussion of Security Properties 26

2.5 Summary and Conclusions 27

3. ADDRESSING VULNERABILITIES IN THE OPERATING SYSTEM 29

3.1 Hardware Monitoring in Low-End IoT Devices 31

3.1.1 Related Work 32
3.1.2 System and Security Model 34

3.1.2.1 Monitoring Graph Construction 34
3.1.2.2 System Architecture 34
3.1.2.3 Security Model 36

3.1.3 Monitor Design 36

3.1.3.1 Operating System Task Management 37
3.1.3.2 Multi-Task Hardware Monitor System 38
3.1.3.3 Context Switch Handling 39
3.1.3.4 Recovery 40

3.1.4 Prototype Implementation 41

3.1.4.1 System Setup and Attack Scenario 41
3.1.4.2 System Operation 42
3.1.4.3 Detection of Attack 42
3.1.4.4 Monitoring Graph for Benchmarks and the OS 43
3.1.4.5 Monitoring System Overhead 43

3.2 Hardware Monitoring in Mid-Range IoT Devices 45
3.3 Summary and Conclusions 46

4. A LIGHTWEIGHT PAYMENT VERIFICATION PROTOCOL FOR BLOCKCHAIN TRANSACTIONS ON IOT DEVICES 47

4.1 Related Work 50
4.2 Horizontally Integrated IoT 51

4.2.1 The Need for Horizontal Integration 52

4.2.2 Economic Relationships and Systems 52

4.3 Current Blockchain-Based Solution 53

4.3.1 Operation 53

4.3.1.1 Setting Up The Joint Account 55
4.3.1.2 Moving Funds 56
4.3.1.3 Closing the Account 56

4.3.2 Drawbacks 57

4.4 The Ticket-Based Verification Protocol 59

4.4.1 Blockchain-Based and Blockchain-Agnostic Operations 59
4.4.2 Separation of Contract Manager and Transaction Verifier 59
4.4.3 Transactions using Tickets in TBVP 61

4.5 Implementation and Evaluation 62

4.5.1 Experiment 1: Sharing Sensor Data 62
4.5.2 Experiment 2: Performance Evaluation 63

4.6 Summary and Conclusion 64

5. SECURING THE EXCHANGE OF DATA AND MONETARY VALUES ON MODERN IOT DEVICES: A MULTI-LAYER APPROACH 66

5.1 Related Work 68
5.2 Implementation of the Ticket-Based Verification Protocol on LEON3 69
5.3 Monitoring the TBVP on a Single-core Processor Running Linux 72

5.3.1 Attacks on Economic Transaction Protocols 72
5.3.2 Attacks on Embedded Systems and Hardware Monitoring 73

5.3.2.1 Security Vulnerabilities in Embedded Systems 73

5.3.3 Hardware Monitor for IoT Protocols 74

5.3.3.1 System Architecture 74
5.3.3.2 Hardware Monitor Operation 75

5.3.4 Hardware Monitor Data Structure and State Transition 77

5.3.4.1 Selective Monitoring 79

5.3.5 Attack Scenario: Stack Smashing Attack 80
5.3.6 Attack Detection and Recovery 81

5.4 Monitoring the TBVP on a Multi-core Processor 83

5.4.1 System Architecture 84
5.4.2 Hardware Monitor Operation and Data Structure 85

5.4.2.1 Task Initiation 85
5.4.2.2 Context Switch 86
5.4.2.3 Process Migration 86

5.4.3 Attack Detection and Recovery 87
5.4.4 Implementation and Evaluation 88

5.4.4.1 Normal Execution on One Core 88
5.4.4.2 Continuous Monitoring In Presence of Task Migration 89
5.4.4.3 Attack Detection and Recovery 91
5.4.4.4 Resource Utilization and Hardware Overhead of the Proposed Hardware Monitor 91
5.4.4.5 Performance Overhead 93

6. CONCLUSION AND FUTURE DIRECTIONS 95

BIBLIOGRAPHY 99

LIST OF TABLES

Table **Page**

2.1 Resource use and power consumption on a Stratix IV FPGA 26

3.1 Qualitative comparison to related work. 34

3.2 Monitoring graph sizes for operating system and applications. 43

3.3 Resource use on a Stratix IV FPGA. 44

4.1 Number of Operations per Second Performed by Each Device in Different Roles in μRaiden versus TBVP. 64

5.1 Related Work on Hardware Monitoring. 69

5.2 Data structure used by hardware monitor 80

5.3 Data structure used by the coordinator 85

5.4 FPGA resources consumed by different entities. 93

5.5 Time spent in the user application versus the kernel when running different workloads for different system configurations. (values are in seconds) 94

LIST OF FIGURES

Figure		Page
1.1	IoT architecture to enable horizontal integration.	6
2.1	System architecture of Multi-Task Hardware Monitor System.	14
2.2	Detailed view of multi-context monitoring system	18
2.3	Console display during stack smashing	23
2.4	SignalTap waveforms showing the trigger for monitor context switch (Operation = 0x2), monitor switch finished (Done), and monitor restart monitoring when CPU ready (Enable)	24
2.5	SignalTap waveforms showing the operations for monitor process creation (Operation = 0x1) and monitor update finished (Done)	24
2.6	SignalTap waveforms showing the successful identification of a stack smashing attack. Instruction hash values are checked against expected values stored in the monitor graph. When a mismatch occurs, a recovery signal is triggered indicating the process should be terminated.	25
3.1	System architecture of embedded hardware monitoring system that can validate correct execution of applications and operating system.	35
3.2	Detailed view of multi-context monitoring system.	37
3.3	Context switch interactions between processor and monitor.	41
3.4	Attack on processor system, which is detected by hardware monitor.	41
4.1	Space-Time diagram for money and data exchange through payment channel as proposed by Poon et al. [54].	54

4.2 Data structure used in the smart contract. 55

4.3 Flow of information and money in (a) current blockchain-based solutions versus (b) our proposed Ticket-Based Verification Protocol 57

4.4 Blockchain transaction details for experiment 1. 61

5.1 Space-Time diagram showing contract setup, data/economic-value exchange, and contract closure phases. 70

5.2 Blockchain transactions for selling 200 data samples. 71

5.3 Proposed IoT system architecture with the monitoring system. 75

5.4 Different states of the hardware monitor and transitions between them. 79

5.5 System console after being hijacked. 81

5.6 (a) Signals extracted from the FPGA board showing the detection of the attack. (b) Resulting blockchain transactions after detecting the attack. 82

5.7 Proposed system architecture for a multi-core LEON3 processor augmented with a hardware monitor. 84

5.8 Waveform extracted from the FPGA showing the Completion of a task on core 1. 88

5.9 Waveform extracted from the FPGA showing the successful transfer of monitoring state across two IVSLs. 90

5.10 Waveform extracted from the FPGA showing the detection of an attack on core 0. 92

CHAPTER 1

INTRODUCTION

Over the past two decades, the idea of Cyber-Physical Systems (CPS) and their use in a broader network, known as the Internet of Things (IoT), has attracted a lot of attention as the technological foundation to address many important societal problems relating to the environment, health care, transportation, etc. [27, 60, 65, 67, 80]. Today, we can see traces of IoT devices in many different aspects of our daily lives. According to Gartner, the number of IoT devices is expected to exceed 25 billion by 2020 [25], which shows how fast this new technology is finding its uses in different applications.

Fundamentally, IoT systems interconnect three types of components: *sensors* that detect or measure a physical property; *computation* that receives sensor data for processing and making control decisions; and *actuators* that act in the physical world in response. These three steps of sensing, computing, and acting can vary vastly in their implementation based on the IoT application and use case.

As the application of IoT systems become more prevalent, new challenges and concerns around their proper operation arise. Among those concerns, we can point to the privacy and integrity of the data throughout the whole cycle, from the sensor to the actuator, and the security of all the devices involved in the cycle with regard to potential external adversaries. In this work, our main focus is on enhancing the security of IoT systems and facilitating secure transactions between them while meeting the privacy requirements of the applications in which they are being used.

Since embedded systems are at the very core of sensing and actuation parts of any IoT system (sometimes even in the computation part), the security and privacy of the whole IoT system relies on the security and privacy of the underlying embedded systems. For that reason, our main focus throughout this work is on embedded systems and challenges around them.

In the first part of this work, we focus on enhancing the security of embedded systems as the underlying processing platform for IoT devices. We discuss the challenges of securing these devices and how to overcome those challenges.

The second part of this work is about the algorithms used at the software level and protocol level to ensure security and privacy of the information being exchanged between different IoT devices. In this part, we describe the idea of horizontal integration, which tries to facilitate the data exchange between different IoT ecosystems to pave the way for further use of IoT solutions in different new applications. As the basis of the idea of horizontal integration is incentivizing data sharing by adding monetary values to the data, having a robust and trustworthy financial mechanism is crucial. We then explain why we believe blockchain technology is a suitable platform for those financial transactions and how the current state of the art methods can be used in that model. At the same time, we discuss how those discussed methods are only applicable to mid-range and high-end embedded devices. As a big portion of IoT devices is based on low-end embedded devices, these approaches come short serving this group of IoT devices. We finally propose our lightweight payment verification protocol, which is based on blockchain technology to make it possible for all of the IoT devices to participate in horizontal integration.

The third part of this work is on joining the two approaches described in the first and second part of the work. In this part, first, we discuss how we plan to expand our hardware monitoring system to be applicable to yet another group of embedded systems which is the high-end embedded systems. We then describe how a secure IoT

ecosystem can be built using a variety of embedded systems including a high-end, a mid-range, and a low-end IoT device, protected by our hardware monitoring approach and running our proposed lightweight payment verification protocol.

1.1 Hardware Monitoring In Embedded Systems

Embedded systems by definition are computing systems that are used in devices with main a functionality other than just computing. The term is usually used as a counterpart to general purpose computers, which are designed solely for computational purposes. Architecturally, embedded processors, which are the core processing elements of embedded systems, are no different than their general purpose counterparts. For example, Motorola 68000 [68], which was introduced in 1979 by Motorola Semiconductors [47], was designed as a high-end microprocessor for computational purposes. However, less than a decade after its introduction, in the early 1980s, it was begin used as an embedded processor inside automobile engines by car designers.

What happened to Motorola 68000 is a good example of how embedded systems' technology follows high performance computing. As the technology matures, processing power can be realized at a lower cost. This cost can be in terms of power consumption, area consumption, and actual monetary cost of production. As in the context of general purpose computers, security has been an active area of research since even before the actual processors came around and since embedded systems are following the footsteps are general purpose computers, one might think why the security of embedded systems should be any different from their predecessors? Why not apply the same techniques that are used in general purpose processors to embedded systems?

The answer to those questions is in the way embedded systems are being used. Although architecturally today's embedded systems are the same as yesterday's general purpose processors, they are different in the tasks they are fulfilling and the

applications in which they are being used. Embedded systems are usually designed on a tight budget. This tight budget usually translates to limited cost, limited energy consumption (in battery operated devices), and limited processing power. These limitations make software-based solutions such as virus detection software and intrusion detection systems which are quite popular solutions in the context general purpose processor, very inefficient for embedded systems.

Another differentiating factor between embedded systems and general purpose computers is the use of them in real-time applications. The real-time nature of those applications paired with the limited processing resources available in the system makes the uncertainty imposed by the software solutions intolerable for these applications. Finally, in some applications such as IoT, the systems are deployed in remote environments which makes accessing them harder. This remote environment is not always a safe environment and in cases might be even hostile. The limited access to the devices and their higher exposure to adversarial parties make software-based solutions which are inherently reactive even less suitable for this type of embedded systems.

An alternative to software-based security solutions is hardware-based solutions. Hardware-based solutions vary from circuit level to architecture and system level. In this work, our focus is on hardware monitors which are hardware solutions at the system level. Hardware monitors are dedicated hardware modules which are located on the same system as the processor. They ideally do not require any changes to the architecture of the CPU and only require certain information extracted from the datapath of the processor. The main role of these hardware modules is to monitor the execution of the application on the processor core and compare that runtime behavior against a golden model of the application. The golden model which is extracted from the source of the application needs to be loaded into the hardware monitor before the application starts to run on the CPU. In the first part of this work, we discuss the details of hardware monitors and why we believe that hardware monitors are

more suitable security solutions than software-based approaches in the context of embedded systems. We then propose our novel hardware monitoring approach which expands the applicability of current hardware monitoring solutions to a wider range of embedded systems including more modern architectures.

1.2 Horizontally Integrated IoT

Deployment of IoT solutions can be expensive since sensors, actuators, and computation components are necessary. Traditionally, each application uses its own components. Switching from dedicated computation infrastructure to cloud-based computation platforms can reduce some of this cost. However, each application still needs to setup and maintain its own set of sensors and actuators in the environment. An alternative approach, which can make this investment more cost effective, is to share that platform with other IoT applications. To facilitate that resource sharing, the idea of *horizontal integration* of IoT components was proposed by Wolf et al. [78, 79].

The system architecture we use in this work is based on the guidelines set in horizontal integrated IoT and is shown in Figure 1.1. The structure aligns with the layered IoT architecture presented by Wolf et al. in [78]. On the top layer, we have the users or applications controlling the system. The user/application needs to formulate and enter the constraints and specifications of the required IoT task. Those constraints and specifications include the functionality of the tasks and how much budget is allocated to this specific task (or how much the cost of using each module in the system is).

Below is the context layer. This layer which receives the constraints and specifications from the user level, forms and enforces the system policies. It is in this layer, where the system decides how much of the resources available in the lower layers are needed to perform the task and how much can be spared. We proposed the *contract management* unit in this layer to deal with leasing of extra resources.

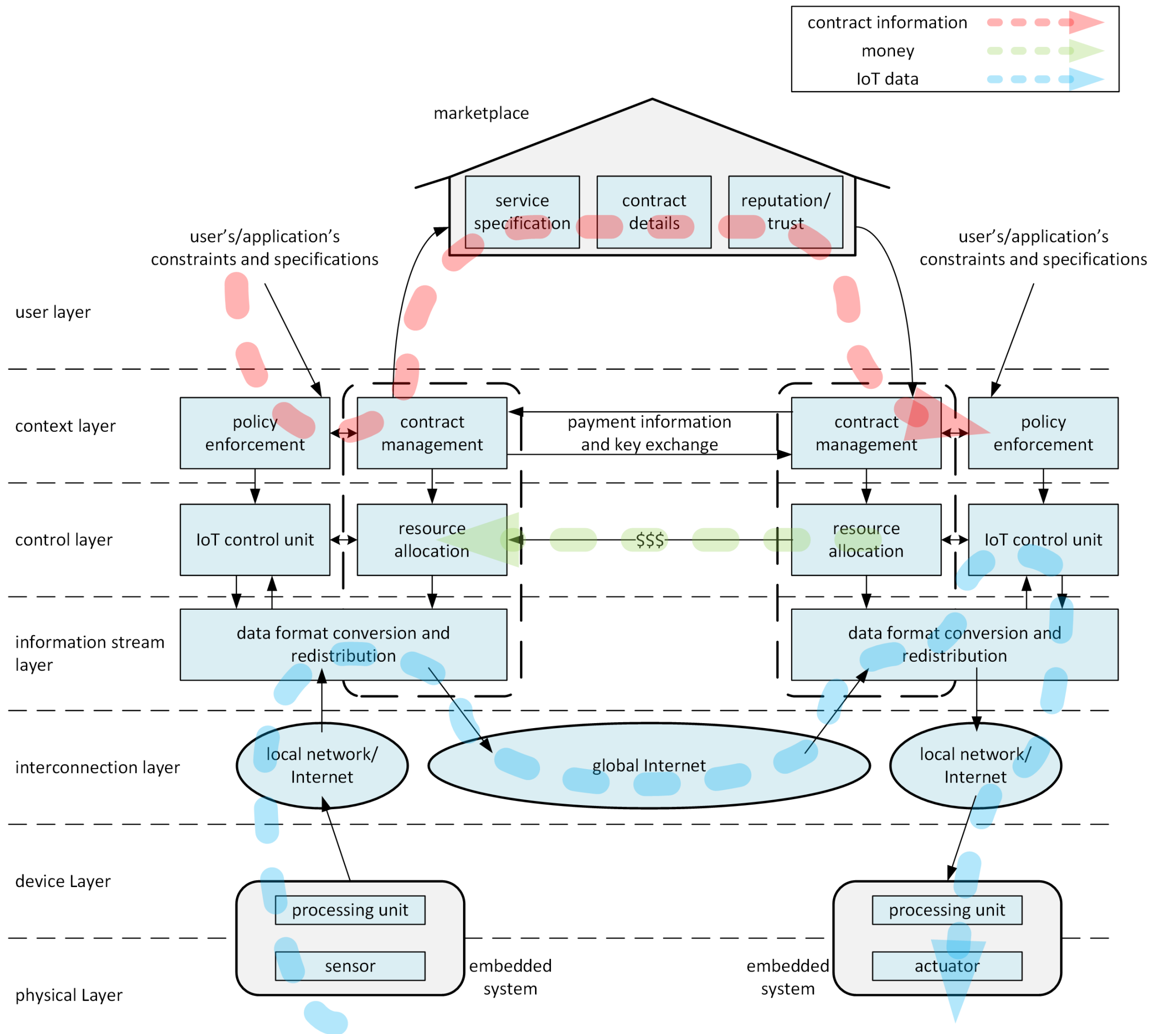


Figure 1.1. IoT architecture to enable horizontal integration.

In case the available resources in the system are not sufficient to perform the task, this unit searches the marketplace to find other systems offering those resources. It then contacts those systems and leases those resources for the period of time they are needed. It is necessary for this unit to be tightly coupled with the *policy enforcement* unit to know about the specifications of the resources needed, the budget dedicated to acquiring them, and finally, about the time period those resources are needed for.

Another role of the contract management unit is to offer the excessive resources the system owns on the market. This role also requires a very close relationship to

the policy enforcement unit as the contract management unit needs to exactly know which resources are idle for what time period and what the price of each resource should be in the market.

When acquiring extra resources, the contract management unit of the lessee inquires about the systems offering those resources on the marketplace. It then selects the one which meets the criteria set by the policy enforcement unit. The contract management unit then directly contacts its counterpart in the lessor system to exchange the information needed to establish a secure connection and also the account information to make future payments. The next step in leasing resources from outside systems is to pass this information to the *resource allocation* unit in the layer below.

Proposed resource allocation unit is co-located with the *IoT control* unit in the control layer. It receives the contract information from the contract management unit and coordinates with the IoT control unit to dedicate the resources accordingly. The resource allocation unit also sets the data format conversion parameters in the layer below and sets up the paths the data from each resource should traverse. This could be a path from the global Internet to the control unit in case of using external resources or it could be a path from the local resources to the information stream layer of another system in case of renting out internal resources. The resource allocation unit is also responsible for sending and receiving money once the contract is formed. By doing so, it makes sure the information channel formed between the two systems is up and running.

1.3 Organization and Contribution

The rest of this work is organized as follows:

In Chapter 2, we discuss the idea of hardware monitoring and challenges of securing embedded systems using hardware monitors in more details. We then describe how the current hardware monitoring solutions have fallen behind of the technology

used in modern embedded systems and in this chapter we take the first step to bridge this gap by introducing our novel hardware monitoring system which can support multiple tasks running on top of a simple realtime operating system.

In Chapter 3, we show how operating systems which previously assumed to be secure entities can be vulnerable against remote attacks and be exploited. Then we show how our monitoring method proposed in Chapter 2 can be further extended to protect the whole system including the operating system itself. It is in this chapter that we discuss how middle-range embedded systems vary from low-end systems in terms of the processor architecture as well as the software running on top of them including the operating systems and how that difference impact the design of our hardware monitoring solution.

In Chapter 4, we shift our focus to horizontal integration of IoT systems and the challenges of securing IoT devices involved in this horizontal integration. In this chapter, we describe how current blockchain technology can be a good fit for this paradigm but at the same time too heavy weight in terms of processing demand to be used in low-end IoT devices. We then propose our lightweight payment verification protocol which is based on Ethereum blockchain technology. We describe how this lightweight protocol can be used even in low-end IoT devices which constitute a big portion of current IoT devices to enables them to horizontally integrate with other IoT systems without jeopardizing their security and privacy.

In Chapter 5, we describe our current plan to further expand our hardware monitoring approach to further cover high-end embedded devices as well as mid-range and low-end ones. We then show how an IoT system can be built using a variety of IoT devices including low-end devices and high-end devices where each of those devices can be secured using the hardware monitor at the hardware level. At the same time, using the protocol proposed in Chapter 4 they can participate in a horizontally integrated IoT ecosystem and securely exchange data and cryptocurrency.

CHAPTER 2

WORKLOAD MONITORING IN EMBEDDED SYSTEMS

Embedded processing systems are widely used and are a key technology for control systems, the Internet of Things, personal health monitoring, home automation, and many other application domains. Due to their wide use and the importance of their tasks, embedded systems need to be protected from hacking attacks. With an increasing number of embedded systems being connected to networks, one typical attack vector against embedded systems is through the global Internet.

Many embedded systems are based on general-purpose processing systems that are vulnerable to the same type of attacks as conventional desktop and server computers, albeit for a different set of applications. The National Vulnerability Database (NVD) [50] shows that around 10% of vulnerabilities (6,518 out of 66,399) in systems are related to overflows that can be exploited via a network. Many of these overflows then enable an attacker to execute malicious code. Thus, the first part of our work, which is presented in this chapter and the following chapter, focuses on protecting embedded systems from this important type of attack using a security-enhanced processor.

While desktop and server computers have the processing power to run malware detection software (e.g., virus scanner, intrusion detection system, etc.), embedded systems are typically not able to do so due to resource constraints (e.g., limited power budget, limited processing capacity, etc.). Instead, hardware-based protection mechanisms have been developed, in particular "hardware monitors", which track the operation of the processing system and aim to detect and suppress malicious activity.

A variety of different hardware-based solutions have been proposed to protect embedded processing systems. In general, there have been three shortcomings in existing work:

- Monitoring on systems with complex workloads is based on coarse indicators (e.g., function call sequence [64]). This approach leaves the system vulnerable to attacks that happen between indicators (e.g., within a function call).
- Fine-grained monitoring systems do not support multi-task workloads on operating systems. This constraint limits the applicability of this single-task monitoring to specialized domains (e.g., embedded control systems, network processors, etc.).
- Almost all of the existing work either target the very elementary embedded systems which do not run an operating system or in case of dealing with the embedded systems running an operating system, the OS is treated as a secure and trusted entity. This leaves the modern OS-based embedded systems vulnerable against attacks which target the operating system itself.

To make hardware monitors an effective protection mechanism for attacks on embedded systems in any application domain, it is critical to develop fine-grained monitoring on multi-task embedded systems. In this chapter, we present the design of a hardware monitoring system that coordinates with the task switching dynamics of an operating system to verify every instruction executed by applications. In the next chapter, we shift our focus to the operating system itself and how to address the vulnerabilities within the operating system.

The specific contributions of our work presented in this chapter are:

- Design of a Multi-Task Hardware Monitor System (MTHM) that supports multi-tasking contexts and that operates in sync with an embedded operating system (OS).

- Prototype implementation of a hardware monitoring system on an FPGA-based DE4 board.
- Evaluation of the prototype and a demonstration of system protection from a stack smashing attack.

This security enhancement for embedded processors allows for the simultaneous use of application-specific monitoring information for multiple applications [73]. The remainder of this chapter describes the design, operation, and implementation of our hardware monitoring system in more detail. In Section 2.1 we discuss other security approaches for embedded processors, including monitoring. Section 2.2 provides the security model and operation of our system while Section 2.3 describes the hardware details and protocols involved in task switching using monitoring. Experimental results are presented in Section 2.4. Section 2.5 concludes the chapter.

2.1 Related Work

Protection mechanisms for processing systems against code injection attacks are manifold. Network devices, such as firewalls [45] and intrusion-detection systems [46], can block malicious network traffic if packet payloads are not encrypted and if detection rules (e.g., Snort [62]) are updated quickly enough. Programming language extensions can generate code that is not vulnerable [37] if source code is available and can be transformed appropriately. Stack protection mechanisms in program code or in the operating system can defend against some attacks [15]. Memory protection mechanisms that separate instruction and data memory (e.g., Harvard architecture or No-eXecute (NX) bit) can avoid some attacks, but are still vulnerable [23]. A survey of these various techniques can be found in the article by Younan et al. [83].

The prevention of stack smashing attacks has been the focus of significant work, although most approaches require significant processor modifications and run-time

execution slowdown. Dynamic instruction flow tracking (DIFT) [69] tags each incoming data value or its derivative with a one-bit tag to indicate that it should not influence program control flow. The approach can require an execution slowdown due to tag checking. CHERI [81] establishes a base and bounds for pointers, preventing illegal accesses to memory which can lead to buffer overflow attacks. This approach also involves data tagging and the use of a special-purpose capability processor and registers to dynamically assess tags. The Hardbound approach [19] includes hardware to check the address bounds of every pointer access to memory. A flexible software-only approach [48] introduces a compiler pass for each application to insert bounds checking operations in the code. Although flexible for a range of applications, an increase in code size and application slowdown make the approach limited for embedded applications.

One very effective protection mechanism is the use of hardware and software monitoring to track different aspects of program behavior. The granularity of such monitors ranges from a call sequence (e.g., [64]) to checksums over basic blocks (e.g., [6]) to per-instruction verification (e.g., [42]). Coarse monitoring granularity may not be able to detect attacks that require only a few instructions to execute (such as demonstrated for a denial-of-service attack in network processors [14]). Thus, our work focuses on monitors that perform per-instruction monitoring and can detect attacks immediately when program behavior changes. Due to the need for tight coupling, such monitors are implemented in hardware and co-located with the processor core.

Existing hardware monitors have been used to monitor processors with single-task workloads (e.g., [42]) or with a small number of tasks that are managed through a control processor (e.g., [32]). However, an increasing number of embedded systems use operating systems, where multiple tasks actively share the processor core and tasks are dynamically added and removed. Our work focuses on providing security

through instruction-level monitoring in such a highly dynamic workload controlled by an operating system.

2.2 System and Security Model

To provide the necessary context for the Multi-Task Hardware Monitor System design presented in Section 2.3, we briefly discuss the operation of MTHM and the security model for our work.

2.2.1 Secure Processing with Hardware Monitors

Hardware monitors are components that are co-located with processor cores to track the processing of software on that core. The objective is to assess the operation of the processor and determine when incorrect behavior is detected (which can be due to benign faults or malicious attacks). As discussed in related work, there are a number of different approaches to monitoring based on what information is communicated from the processor to the monitor and what information is used to determine if that behavior is "normal".

In our work, we use a hardware monitor that receives information about every instruction executed on the processor core and compares it to a "monitoring graph" that is based on the analysis of the processing binary (similar to [42]). Each instruction is represented by a 4-bit hash value (to reduce the size of the monitoring graph compared to the size of the binary) and state transitions correspond to possible control flow paths between instructions. We use a deterministic finite automaton (DFA) representation of the monitoring graph (as described in [12]).

The system architecture of our Multi-Task Hardware Monitor System, which supports multiple tasks, is illustrated in Figure 2.1. The figure shows that application binaries are analyzed offline. During runtime, the comparison logic in MTHM matches the monitoring graph to the currently active task on the processor. To do the

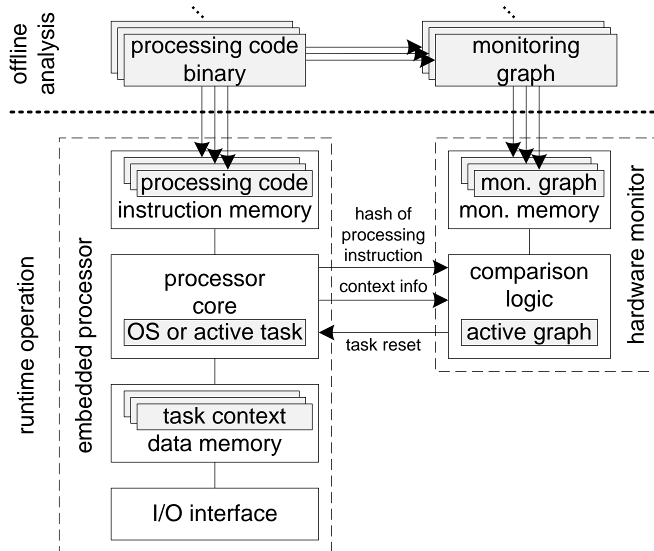


Figure 2.1. System architecture of Multi-Task Hardware Monitor System.

operation, the OS-to-Monitor Interface (OMI) communicates the necessary context information between the processor and the monitor. When the processor execution does not match the expected behavior reflected in the monitoring graph of the current task, a recovery signal is sent from the monitor to the processor to terminate the current task. (More complex recovery and roll-back mechanisms could be implemented, but are not discussed here.)

It is important to note that the hardware monitoring system is isolated from the processor and thus cannot be tampered with remotely by the attacker (e.g., to change the monitoring graph to match an attack). Related work discusses how to achieve such isolation while still enabling dynamic installation of hardware monitoring graphs through the use of cryptographic mechanisms [34].

2.2.2 Security Model

To justify how our proposed system provides a secure processing environment, we briefly discuss the security model that is the basis for our work.

2.2.2.1 Security Requirements

We require that our system meets the following security requirements:

SC1 The system should only allow execution of code as programmed in the executable binaries of each task.

SC2 Secure processing should be provided for multiple, dynamically changing tasks.

SC3 Malicious code execution in one task should not affect other tasks.

In addition to security, there are also practical performance requirements. As we show in our results, the hardware monitor does not reduce the performance of the embedded processor in any way. The only overhead is a few instructions (five for our experimentation) in the operating system code when switching tasks, which leads to a negligible reduction in processing speed.

2.2.2.2 Attacker Capabilities

We make the following assumptions about the capabilities of an attacker that tries to change the operation of the embedded system and/or tries to execute malicious code on the embedded system:

AC1 An attacker can provide any input through input/output interfaces of the embedded system.

AC2 An attacker can start and stop any task from an installed binary in the embedded system (within the limitations of a maximum number of active tasks).

AC3 An attacker can tamper with any of the binaries.

In order to provide a practical solution for secure processing in an embedded system, we also require some reasonable constraints on attacker capabilities:

AC4 An attacker cannot tamper with the operating system itself.

AC5 An attacker cannot tamper with the hardware monitoring system (e.g., modifying monitoring graphs for installed executables).

As discussed above, we do not discuss the secure installation of monitoring graphs, which has been addressed in related work [34] in more detail.

2.3 Monitor Design

2.3.1 Task Management in the Operating System

A key aspect of our monitoring system is its ability to fit seamlessly within the context switch operations of a typical operating system. As noted in Section 2.4, the time required to switch monitoring graphs for different tasks is significantly less than the typical time required for other activities in a context switch. In our implementation, graph switching is synchronized with other OS actions (e.g., register file save and restore) that occur during a context switch so that user tasks are protected at all times. Typical context switch activities for embedded operating systems, such as μC/OS-II[1] used for this work, include:

1. A timer or other OS event generates an interrupt triggering a context switch.
2. The OS scheduler determines the next process for execution. Our implementation uses a priority based scheme, although it can be replaced with a round-robin or any other scheduling algorithm if necessary.
3. The OS provides the process ID (PID) of the next process to the monitoring system, triggering a monitoring graph switch in the monitor. This switch includes monitor state saving for the process currently being monitored, and a restoration of monitoring state for the next process.

[1] http://micrium.com/rtos/ucosii/overview/

4. Concurrently, the OS saves process state (registers, program counter, etc.) for the current task to main memory.

5. The OS retrieves process state for the next process from main memory and restores it to processor registers.

6. The OS checks the status of the monitoring system to confirm that the monitor for the next process is ready for use.

7. The OS sends a trigger to the monitoring system to start monitoring for the newly-loaded process.

After the context switch is completed, the processor sends every instruction executed for the process to the monitoring system. In the next section, we provide a detailed view of the monitoring system and how it interacts with the processor for steps 3, 5, and 6 above.

2.3.2 Multi-Task Hardware Monitor System

A detailed view of our monitoring subsystem is shown in Figure 2.2. The portions of the monitoring system can be split into three parts: *monitoring hardware* (three boxes in upper left corner of the figure), which checks the per-instruction operation of the companion processor; *graph memory*, which stores state information about monitoring for each process, *controller*, and the *processor interface.*

The monitoring hardware checks each processor instruction using information from the monitoring graphs stored in graph memory. In the figure, graphs for four separate applications are stored in *slots* in the graph memory. Each graph includes one row per instruction, effectively representing expected program control flow as a state machine [12]. A *read address* pointer indicates the entry in the graph that corresponds to the instruction that has just completed execution. During the execution of an instruction, a multi-bit (in our case 4-bit) hash value of the instruction is generated and converted

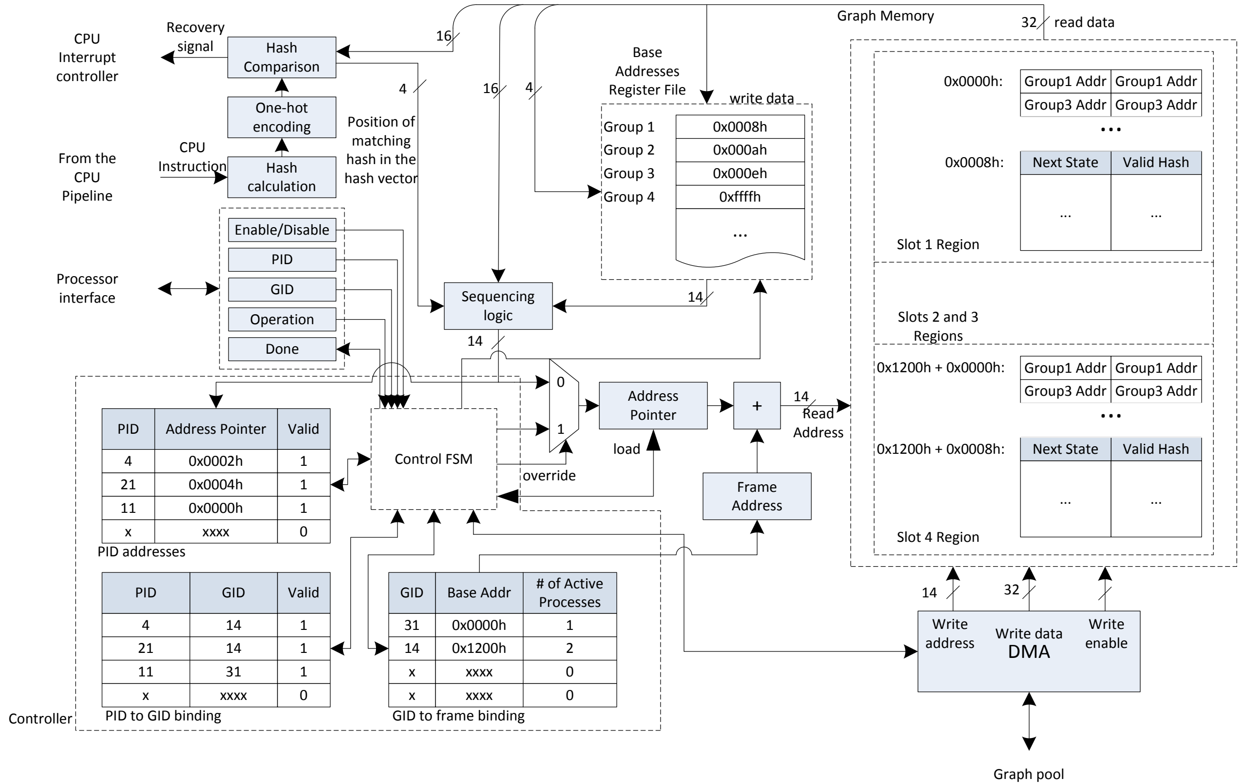


Figure 2.2. Detailed view of multi-context monitoring system

to a one-hot representation. This one-hot encoding is compared against the expected next-instruction hash values (*valid hash*) that are stored in the graph entry for the previously executed instruction. Since branch instructions may have several possible next instructions, and, consequently, several possible valid hashes, multiple one-hot valid hash bits may be set per entry. A match of any of these hashes indicates a valid instruction. If no match occurs, an illegal instruction has been executed, leading to the generation of a recovery signal which alarms the processor and prevent the attack code from spreading or affecting the rest of the ecosystem. Our approach can handle dynamic branch targets by profiling the code to determine all branch targets for an application prior to graph generation. Entries for these targets are then added to the graph.

In the memory graph, states are grouped based on their fanin count and are placed in the memory with regard to their groups (i.e. states within the same group are located next to each other). *base address registers* cointains the base address of the group to which the current state belongs [12]. The next *read address* (memory row) in the monitoring graph is determined using next state information stored in the current entry, the matched hash value, and information stored in *base address registers.* These values are combined via addition and multiplication in the *sequencing logic* box in the figure. The resulting address is stored in the *address pointer* and subsequently added to the start address for the appropriate graph slot for the application. The implemented monitor requires only one memory lookup per instruction.

Effectively, the monitoring information for each process at any given point in execution is defined by the contents of the *address pointer*, the monitoring graph for the process and the contents of the *base address registers.* If a context switch is requested, these values must be updated to use values for the requested next process. The procedure required for a context switch inside the monitoring system is described next.

2.3.3 OS-to-Monitor Interface for Context Switch

In case of a context switch, control information is exchanged between the processor and the monitoring system. The exchange of monitoring information (Step 3 in Section 2.3.1) starts when the processor writes the *PID* of the next process into the *PID* register in the *processor interface* of the monitoring system and sets a bit in the *Operation* register. The monitoring system *control FSM* then performs the following actions:

1. The *address pointer* for the currently executing process is saved in the *PID addresses* storage so that it can be restored for the next invocation of the process.

2. The graph ID (*GID*) associated with the next process is located in the *PID to GID binding* storage using the *PID* written to the processor interface.

3. If the graph ID of the next process differs from the ID of the previous one, the *base address registers* are loaded with values for the graph of the next process. These values are loaded from the graph memory (e.g., Group 1 Addr, etc).

4. The *GID* is used to determine the *frame address* for the start of the appropriate monitoring graph in graph memory for the process. This information is stored in the *GID to frame binding* storage.

5. The *address pointer* value for the next process is restored from the *PID addresses* storage.

6. The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.

7. Once all other context switch activity for the next process has concluded (e.g., processor registers are loaded), the processor sets an *Enable* bit in the *Operation* register of the processor interface, resuming monitoring. The processor waits until this bit set is successfully made, ensuring synchronization. Instructions of the newly-loaded process are then monitored.

In Section 2.4 we show that these steps can be performed in 17 clock cycles for our prototype system.

2.3.4 OS-to-Monitor Interface for Process Creation

When a new task is being created by the OS, it is assigned a unique *PID* and a *GID* by the operating system. Each known application has a predefined *GID* which is known to both the processor and hardware monitor. Since many processes of the

same application may exist, the *GID* may not be unique. The following steps are used to initialize the security monitor for the new process.

1. The two identifiers (GID and PID) are passed to the monitor via the *processor interface.* The monitor first searches for an empty slot in the *PID addresses* storage and *PID to GID binding* storage to insert the new bindings.

2. While making these associations, the *GID to frame binding* storage is searched to determine if the appropriate graph is already loaded. If it is available, the next step is skipped.

3. If the *GID* is not found in the *GID to frame binding* storage, the *GID* is inserted into the table. After a graph to remove is determined using a least recently used approach, the new graph is then loaded into graph memory using the DMA interface. Following graph loading, base addresses are updated.

4. The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.

During system startup, monitoring graphs are loaded from an external memory graph pool for the new processes that will be executed by the processor. Concurrently, the processor performs a series of process creation operations including initialization of the process stack and control block (registers, etc.). In Section 2.4, it is noted that while process creation can require hundreds of cycles for the processor, if the appropriate monitoring graph is already in the monitoring system, monitoring information update for process creation requires less than 20 cycles for the monitoring system.

2.4 Prototype Implementation

2.4.1 System Setup

To verify the functionality of our monitoring system, we implemented an embedded NIOS II-based processing system plus monitoring system using a Stratix IV GX230 FPGA located on an Altera DE4 board. A single-core NIOS executing a μC/OS-II operating system was used for testing. Monitoring logic and memory were implemented in on-chip resources. Monitoring graphs were generated by passing code through a standard MIPS_GCC compiler flow to generate assembly-level instructions [12]. The output of the compiler allows for the identification of branch instructions and their target addresses. This information was used to generate monitoring graphs for four MiBench[2] applications (*bitcount*, *qsort*, *basicmath*, and *stringsearch*) and malicious stack-smashing attack code. Our examination of all MiBench benchmarks determined that the target for all dynamic branches could be determined at compile time.

The attack code we use for our system is a C function which accepts a character string as its only argument and copies it to a buffer located on its local stack frame [66].

```
void process_input(char *stringpassed) {
    char name[90];
    strcpy(name,stringpassed);
    printf("Processing string .. !\n");
    return;
}
```

In this poorly designed code, no check is made to determine if the string **stringpassed** is longer than the target buffer, so the return address of the function can be overwritten with an address which points into the user-provided input string. Instead

[2] http://wwweb.eecs.umich.edu/mibench/

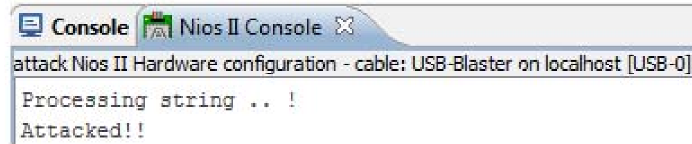


Figure 2.3. Console display during stack smashing

of characters, this "string" can contain processor instructions which repetitively print out "Attacked!!" on a terminal in a loop, although much more malicious behavior could be imagined. A monitor for the code is able to detect the unplanned control flow jump and stop the processor before the attack can perform this activity. The hash values stored in the monitoring graph for the application will not match the values for the malicious instructions as they are executed during the attack. As shown in Figure 2.3, we have confirmed that this attack will lead to unexpected results (an attack message) if monitoring is not used.

2.4.2 Monitor Context Management

We have verified our ability to perform numerous context switches between multiple processes of the four monitored MiBench benchmarks both via simulation and in emulation hardware. This switch includes both standard process state used by the processor (e.g., register information, stack) and monitoring information using the mechanism outlined in Section 2.3.3. Altera SignalTap, a hardware debugger, was used to generate the waveforms shown in Figure 2.4.

The waveforms show the synchronization between the processor and the monitor as a result of the context switch. First, the processor notifies the monitoring system of the switch by writing the *PID* of the next process into the processor interface. The monitor switch is started by the processor writing into the *Operation* register of the interface. The value of the *address pointer* for the old process is stored and the value for the new process is restored to/from *PID* address storage immediately

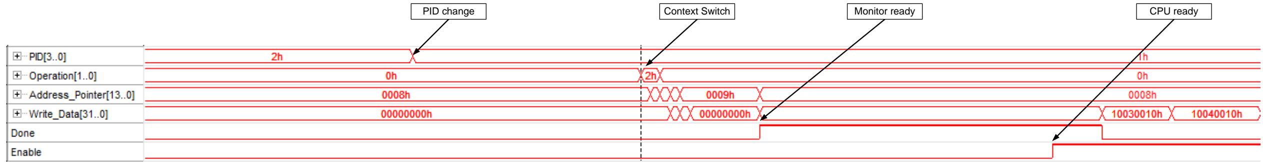


Figure 2.4. SignalTap waveforms showing the trigger for monitor context switch (Operation = 0x2), monitor switch finished (Done), and monitor restart monitoring when CPU ready (Enable)

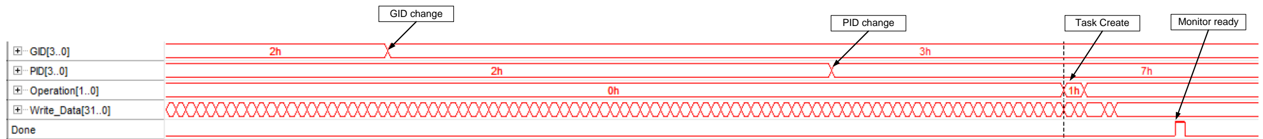


Figure 2.5. SignalTap waveforms showing the operations for monitor process creation (Operation = 0x1) and monitor update finished (Done)

after this trigger. The base address registers are then configured using the *write_data* port shown in Figure 2.2. After the control FSM performs the monitor update, the *Done* signal is set in the processor interface indicating the monitor context switch is finished. Finally, after the processor finishes other context switch operations, it sets the *Enable* signal in the processor interface to restart monitoring. The processor waits a cycle until this write is complete. Monitoring for the new process starts with the first instruction received from the process.

Experiments in simulation and in the lab on FPGA hardware showed that the processor is able to process data for the MiBench benchmarks equally fast both with and without monitoring (e.g., no slowdown for monitoring). Context switch time is extended by 5 cycles versus no monitoring to allow for monitor context switches. This overhead accounts for the data exchanges between the processor and monitoring system for synchronization. Overall, we found that the number of cycles needed to perform a monitor context switch is 17 versus the 34 cycles needed for the processor to

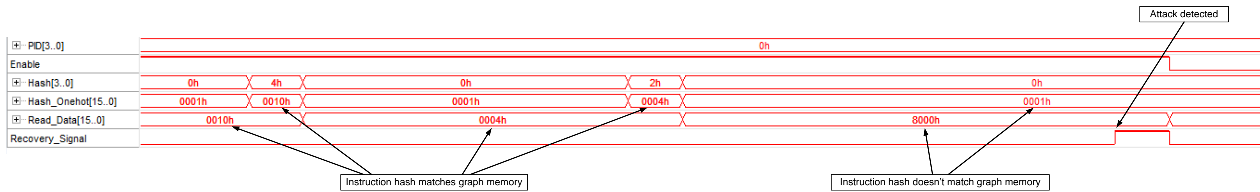


Figure 2.6. SignalTap waveforms showing the successful identification of a stack smashing attack. Instruction hash values are checked against expected values stored in the monitor graph. When a mismatch occurs, a recovery signal is triggered indicating the process should be terminated.

save and restore registers (note that monitor and processor context switch operations occur in parallel).

The amount of time needed to create a new process in the OS is about 600 clock cycles versus 17 to create process information in the monitor (Figure 2.5). If a monitoring graph is loaded from main memory, the cycle count required for the monitor increases to include reading the number of rows in the monitoring graph for the new process into graph memory (about 80 for each of our applications).

2.4.3 Attack Detection and Protection

We have verified in both simulation and in hardware that our monitoring system is able to detect the stack smashing attack described in Section 2.4.1 and notify the processor so that the malicious process can be terminated. SignalTap waveforms derived from observing hardware operation in system are shown in Figure 2.6. As described in Section 2.3.2, an attack is detected when the hash of the CPU instruction does not match the expected value stored in the monitoring graph for the application. In our system, the implemented hash function counts the number of ones in the instruction to form a four-bit hash value. The figure shows the four-bit hash value, a one-hot version of the hash value, and the retrieved, expected hash value for the instruction from the monitoring graph (*read_data[15:0]*). In the waveforms, it can

	Available on FPGA	Nios II with no HW monitor	HW monitor and controller
LUTs	182,400	1,341	406
FFs	182,400	1,166	522
Mem. bits	14,625,792	2,108,416	524,512
Pwr (mW)	-	105.97	41.83

Table 2.1. Resource use and power consumption on a Stratix IV FPGA

be seen that the correct hash value is matched twice, but the third hash value is incorrect, indicating a branch to an unexpected section of code. As a result of this detection, a recovery signal is generated, notifying the processor that the process should be terminated.

2.4.4 Monitoring System Resources

To provide some context regarding the amount of overhead required by the monitoring system relative to the processor, hardware results of the system reported by the Altera Quartus II tool are shown in Table 2.1. The lookup table (LUT), flip flop (FF), and memory resources required for the monitor are appropriate compared to the processor core. Dynamic power values are also shown in the table. These power numbers were generated using Altera PowerPlay.

2.4.5 Discussion of Security Properties

We argue that the system we have designed and prototyped achieves the security requirements we put forth in Section 2.2.2.

The key observation is that our hardware monitor can detect when a specific task executes code that is different from the binary. In such a case, the hash value that is reported from the processor core to the monitor does not match. There is a chance that the attacker is lucky and the hash matches by coincidence or the attacker is clever and aims to construct code that matches. This action, however, is very difficult to achieve in practice and can be defeated by hiding the hash function [34]. In Chapter

3, we introduce a technique to add diversity to hardware monitoring systems in a way that even a successful attack to a particular application on one system would fail on other systems. If the monitor detects deviation from the binary, then the processor is signaled to stop execution of the attacked task. Thus, SC1 (no execution of attack code) is achieved.

Our system supports multiple tasks that are switched dynamically by the operating system. The hardware monitor follows along in sync and associates the current task on the processor core with the correct monitoring graph. Thus, we achieve SC2 (secure processing for multiple tasks).

Finally, when an attack occurs, the hardware monitor informs the operating system about the attack and the targeted tasks are stopped using a conventional task termination mechanism (similar to the `kill` command). This mechanism is specifically designed to not affect other tasks. Thus, SC3 (isolation of attacked task) is achieved.

We rely on the limitations of attacker capabilities, such as AC4 and AC5 (no tampering of operating system or hardware monitor), to ensure that an attacker cannot circumvent the security mechanisms we have put in place.

2.5 Summary and Conclusions

In this chapter, we presented security hardware for embedded processors that execute multiple processes under the control of an operating system. Our monitoring approach allows the operation of each process to be tracked at the instruction execution level. If a deviation from the expected instruction execution sequence is detected, the monitor can quickly identify it and notify the processor to initiate process termination.

A significant contribution of the work is the inclusion of multi-context support in the monitoring system. Monitoring state for each process can be quickly saved during

a process context switch and previously-stored state can be reloaded. We document the specific steps needed to ensure synchronization between the processor and monitor to ensure that each process is always protected during execution. Using prototyping, we show that our system is effective for multiple processes managed by an embedded OS. A stack smashing attack is identified and suppressed. The monitoring system does not impact application execution time.

In this chapter, we assumed that the operating system is a secure entity and the main vulnerable point of an embedded system, is the applications running on top of that operating system. In Chapter 3, we revisit this basic assumption that considers the operating systems secure. In two different scenarios, we show how both a real-time embedded operating system such as μC/OS-II and a full-scale general purpose operating system such as Linux can also be vulnerable against attacks. We also expand our hardware monitoring system accordingly in each scenario to ensure the security of the system in presence of external threats targeting those vulnerabilities.

CHAPTER 3

ADDRESSING VULNERABILITIES IN THE OPERATING SYSTEM

The Internet of Things (IoT) represents the convergence of cyber-physical systems (CPS), which control physical processes, and the Internet, which provides global interconnectivity for access to data systems. Embedded systems are at the core of any IoT solution as they provide the necessary computational power at the location where devices interact with the physical world. Due to their deployment in the environment, these embedded systems are typically constrained in their computational resources (performance and/or energy) but still connected to a network to interact with the other components of the IoT solution.

There are three aspects that are of particular importance when talking about the security of IoT devices. First, the IoT devices are connected to the Internet and thus vulnerable to remote attacks. Second, these embedded systems typically do not have the processing capacity or power budget to run software-based defense mechanisms, such as virus scanners or intrusion detection systems. Such software-based solutions, although very commonly used as security solutions in network-connected workstation and server computers, do not seem very suitable for IoT devices. Third, despite their limited processing capability, the collective power of IoT devices is enormous due to the big number of instances deployed in different environments and for a variety of applications. Therefore, in case they go rouge, the implications could be catastrophic.

As discussed in Chapter 2, an effective defense mechanism that has been developed in related work is “hardware monitors”. These monitors are logic components that

are co-located with the embedded system processor core and track the execution of software. Hardware monitors require no change or addition to the software that is run on the processing system.

Hardware monitors were initially introduced to protect embedded applications running on top of a bare metal or simple firmware-based embedded system. However, given the ever-changing nature of IoT applications and the recent advancements in the technology used in IoT devices, the assumption of having a single embedded application running on bare metal or a simple firmware does not fit the characteristics of these modern devices anymore.

In an attempt to bridge the gap between the security requirements of IoT devices and the available security solutions, in Chapter 2, we introduced a novel hardware monitoring system which had the capability of monitoring multiple tasks running on top of an embedded operating system. We showed how that monitoring system was able to stay in sync with the processor core throughout the core's entire processing cycle and during the context-related operations such as process creation, context switching, and process termination. We finally showed the effectiveness of this approach in defeating a buffer overflow attack in practice, using an FPGA prototype of the proposed method alongside an embedded processing system.

In this chapter, we further expand our proposed hardware monitoring approach to accommodate to yet another security need of today's IoT devices: the security of the system amid the vulnerabilities within the operating systems managing these devices. In This chapter we explore the challenges of securing a low-end IoT device managed by a simple real-time operating system. As an example, we use a NIOS II-based embedded system running μC/OS-II. We show how we can extend our hardware monitoring solution to not only protect the system while running user applications, but also continuously monitor the system while the processor switches from a user task

to the operating system or vice versa, and throughout the execution of the operating system itself.

We also give a brief introduction to the work published by Provelengios et al [58] which is a further advancement of the hardware monitoring technique to expand the applicability of this technique to more advanced embedded systems running a Linux kernel instead of a lightweight embedded operating system. We also briefly talk about our future plans to extend this work to accommodate to the security needs of high-end IoT devices in Chapter 5.

3.1 Hardware Monitoring in Low-End IoT Devices

Observing the recent advancements in the microprocessor technology and the trend of using microprocessors in IoT devices, it is safe to say that single tasked application specific processors have no use in today's IoT systems. Microcontrollers with single or multiple cores running an operating system are becoming mainstream. For example, Raspberry Pi Zero [61] which has a single-core Arm 11 [4] processor and can run a full-scale Linux, can be bought for 5 USD. As a result, the old hardware monitoring approaches which target simple embedded applications running on bare metal are no longer applicable to these new IoT devices.

In Chapter 2, we took the first step to address the challenge of securing these new embedded systems by introducing multi-task support to the current hardware monitoring solutions. In this part, we are going to expand that monitoring system to include the real-time operating system as well.

In this part, we present the expanded version of our hardware-based monitoring system that can track each instruction that is executed by the embedded processor and check if it matches the expected behavior of the system. To determine what behavior is correct, we analyze the operating system (OS) and application binaries to create a *monitoring graph* for each. If the system is attacked, it necessarily will execute

instructions that are not part of such a monitoring graph and thus the hardware monitor can detect this deviation. Our system is able to track the dynamics of the system (e.g., context switches and operating system interrupts) to ensure that the monitor can verify the faithful execution of every single instruction on the embedded processor.

The main contribution of this first part of our work is a lightweight security mechanism that can track operating system and application execution and detect attacks at the granularity of individual processor instructions without needing to know any characteristics of such an attack. Specifically, this chapter presents the following contributions:

- Design of a hardware-based monitoring system that can detect any deviation in processing behavior in the operating system or application tasks, even when caused by previously unknown attacks.

- Prototype implementation of a hardware-based monitoring system on an Altera DE4 FPGA board using the μC/OS-II operating system to show the feasibility of this approach.

- Evaluation of the prototype system shows the ability to dynamically switch contexts, handle interrupts, and detect attacks while requiring only a few hundred logic gates and memory comparable to that of the instruction code and causing a minimal processing slowdown of 6 processor cycles per context switch.

3.1.1 Related Work

The importance of security in embedded environments, such as in IoT, has long been acknowledged in academic research [43] and by government institutions [22]. Recent attacks on Internet infrastructure exploited vulnerabilities in IoT devices to launch distributed denial-of-service attacks [63], which highlights the continued need for novel security solutions in this space.

Several techniques are used to provide operating system security at run-time. Typical mechanisms include a trusted computing base and a reference monitor [36]. The software mechanisms enforce a security policy and access to compute objects, respectively. Dynamic information flow security [70] can be applied to operating systems to prevent data from input channels from being used as instructions or jump targets. A data-centric approach adds security information to storage locations and registers to track security levels [75]. A more recent approach uses a neural network to evaluate use patterns for the processor program counter and cycles per instruction [84]. Anomalous operating system behavior can be observed from these parameters. The Tamper Evident Processor [76] tags data values with hashes to identify unexpected changes. These values are used to identify OS data modifications.

The idea of using a hardware-based monitoring system to detect processing deviation is certainly not new: Monitors have been used to track function and system call sequences (e.g., [64]), to verify checksums over basic blocks (e.g., [6]), and to validate execution at a per-instruction level (e.g., [41]). What is new in our work is that we extend our instruction-by-instruction level hardware monitoring approach for an environment of complex, interacting software components (i.e., multiple processing tasks running on an operating system). Such a fine-grained monitoring approach has not previously been demonstrated for a full-scale operating system with multiple tasks. Our previous work in [73] and the work presented in [31] consider multiple processing tasks but do not monitor the operating system itself, which is often the target of attacks. Coarser-grained approaches have considered operating systems and processing tasks, but do not track processing behavior at the level of individual instructions, which opens them up to vulnerabilities, such as described in [14], where attacks have been executed on network processors using only a few instructions of malicious code. A qualitative comparison of related work and our contribution is shown in Table 3.1.

Table 3.1. Qualitative comparison to related work.

	malware scanner	Arora et al [6]	Mao et al [41]	Hu et al. [31]	Our previous work [73]	this work
technique	software	hardware				
overhead	high	low				
granularity	I/O	basic block	processor instruction			
programs	multiple	single		multiple		
OS support	yes	no			yes	
OS monitor	yes	no				yes

3.1.2 System and Security Model

To provide context for our design that we describe in Section 3.1.3 and evaluate in Section 3.1.4, we briefly discuss the system architecture, the construction of a monitoring graph, and the security model that is the basis for our work.

3.1.2.1 Monitoring Graph Construction

The basic idea of hardware monitoring is to compare system behavior against a golden indicator. Here we use a fine-grained indicator called a monitoring graph. This graph is a deterministic finite automaton in which the states are the assembly instructions and edges are the possible transitions between those instructions. It can be constructed by analyzing the binary code of an application. A big challenge in graph construction is resolving indirect control flow instructions where the target address of an instruction is determined by the content of a register. To handle these instructions, source code analysis, profiling, and binary code emulation [72] can be used. The graph extraction process is discussed in detail in [13].

3.1.2.2 System Architecture

The system architecture of our hardware monitoring system is shown in Fig. 3.1. The processor core reports each executed instruction to the hardware monitor which compares the instruction against an entry in the monitoring graph. In the case of

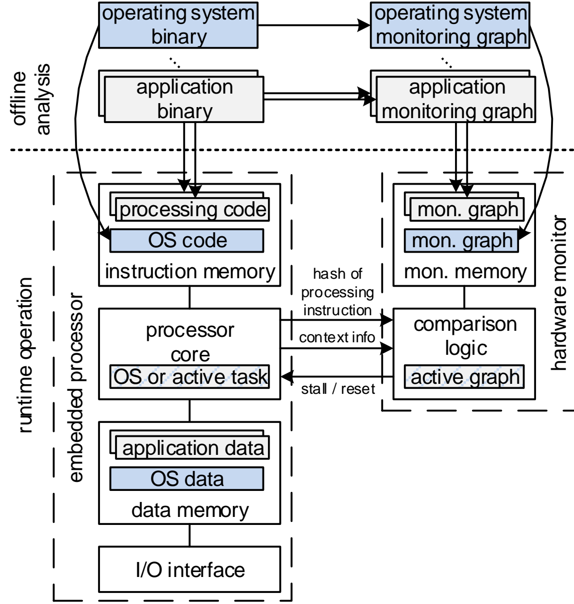


Figure 3.1. System architecture of embedded hardware monitoring system that can validate correct execution of applications and operating system.

deviation for an expected graph traversal due to attack, a recovery procedure is initiated, as discussed in Section 3.1.4.

Such a hardware monitoring approach has been described in related work [6,41,64]. We choose the per-instruction monitoring approach proposed in [41], rather than the per-basic-block monitoring presented in [6] or the system-call monitoring presented in [64], to ensure fast response to attacks. Also, we use the 4-bit hash of the processed instruction for reporting described in [34] (rather than the full instruction word) to reduce memory requirements. Finally, we use the security techniques described in [34] to prevent an attacker from tampering with the monitoring graph to avoid detection.

The challenge for this system, which is the main novelty over related work, is the need to associate the current processing context (operating system or one of multiple applications) with the correct monitoring context. As illustrated in Fig. 3.1, each application and the OS have their own monitoring graph (and associated monitoring state). When the processor switches between application and operating system pro-

cessing (e.g., due to context switch, interrupts, etc.), the hardware monitor needs to follow along with these dynamic changes.

3.1.2.3 Security Model

Having described the system architecture and the construction of a monitoring graph, we consider security properties that are tied to the security model. The security requirements of our system are:

• SR1: The system should only execute code that belongs to the operating system or any of the validly installed applications.

• SR2: Any attack that introduces malicious code should be detected and stopped.

The attacker capabilities that we assume are:

• AC1: An attacker has access to the embedded system through any input/output channel.

• AC2: An attacker can tamper with application and operating system binaries loaded into the main memory, the processing stack, and data memory.

In our work, we also assume the following limitation on attacker capabilities:

• AC3: An attacker cannot tamper with the monitoring graph (e.g., by using the techniques from [34]).

In addition to these security requirements, there are performance requirements, such as low implementation cost and low performance overhead, to make the system practically useful. The hardware monitoring system described in the following section meets these requirements as we show in Section 2.4.

3.1.3 Monitor Design

This section describes the various aspects of our hardware monitor design in detail.

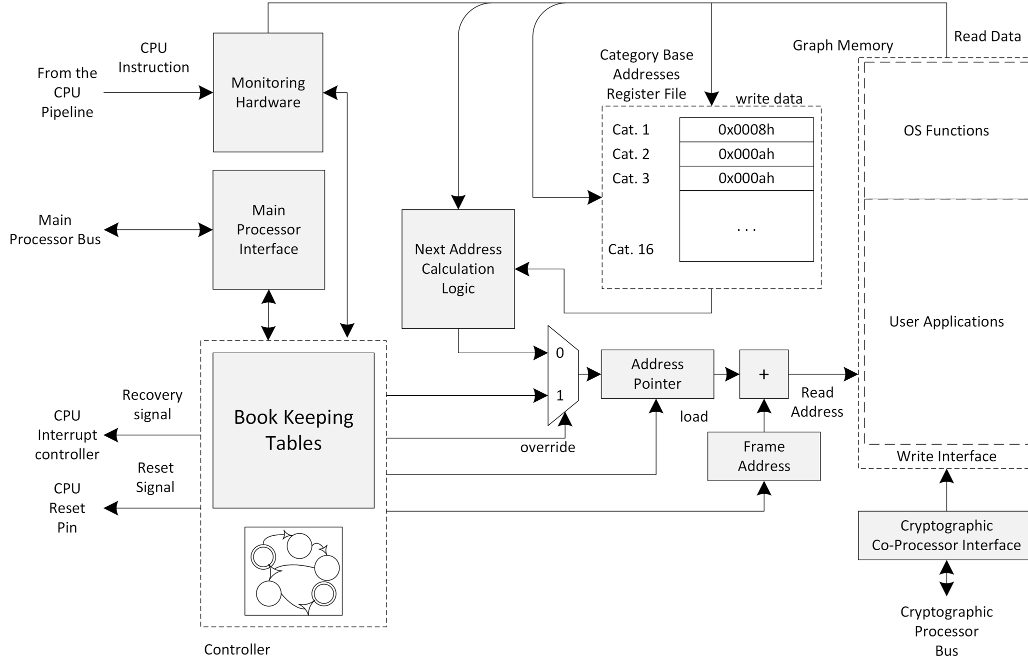


Figure 3.2. Detailed view of multi-context monitoring system.

3.1.3.1 Operating System Task Management

The key aspect of our monitoring system is its ability to monitor both user and operating system tasks. Task switching can be initiated by the operating system (e.g., new user task is scheduled), by applications invoking system calls, or by external events (e.g., timer or external interrupt). In the first two cases, the context switch happens synchronously, meaning that the instruction after which the context switch occurs is known. Therefore, the appropriate information can be provided to the hardware monitor prior to the event by adding a small piece of code to the OS and the applications. However, in the latter case, the context switch can happen with no prior notice.

Context switch procedure on the hardware monitor includes saving the state of the monitoring graph for the code currently being executed and switching to the graph for the next processor task. Since interrupts can happen asynchronously, this whole procedure should be done seamlessly and without any coordination between the main

processor and the hardware monitor. It also must be ensured that monitoring is synchronized with OS task execution following OS context switch operations such as register file save and restore.

Our OS monitoring approach has been developed for μC/OS-II, a widely used embedded operating system. All the OS internal functions (e.g. task scheduling), interrupt service routines, and system calls, handling software traps were continuously monitored.

3.1.3.2 Multi-Task Hardware Monitor System

A detailed view of our monitoring subsystem is shown in Fig. 3.2. The portions of the monitoring system can be split into: *monitoring hardware*, which checks the per-instruction operation of the companion processor; *graph memory*, which stores state information about monitoring graphs; *sequencing logic*, which determines the next state in the graph; *processor interfaces*, which coordinates with the processor when context-related information is received; and *bookkeeping tables*, which associate monitoring graphs with specific user and OS tasks. The tables also keep track of the monitoring status for each graph. Monitoring graphs can be loaded into a secure memory from external memory via a cryptographic coprocessor. Activity in the monitoring hardware is controlled by a finite state machine. We have implemented this system and evaluated its performance on a DE4 FPGA board.

For each executed instruction, the monitoring hardware checks the instruction versus an entry in the associated monitoring graph. If an unexpected result is determined from the comparison, the instruction execution is flagged as a possible attack and the processor is either reset or interrupted. Graphs are loaded into *slots* in the graph memory. Once loaded, the starting address of the slot is associated with the graph ID (GID) of the appropriate graph in the bookkeeping tables.

Once the OS creates a new task, it sends a message to the hardware monitor with the process ID (PID) of the newly created task and its relevant graph ID. Then the hardware monitor loads the appropriate graph into its graph memory if it is not already resident and associates this PID with the received GID in the bookkeeping tables. These bookkeeping tables are consulted and updated during a context switch.

3.1.3.3 Context Switch Handling

In the processor, context switches can be triggered by three different events: *Interrupts, System Calls*, and the *Scheduler* resuming a user application. Next, we discuss how the hardware monitor follows the processor's context switch in each case and thus ensures the security of the system continuously.

• Interrupts: The most frequent triggers of context switching are *interrupts*. Since interrupts happen asynchronously, the physical interrupt signal (IRQ) is presented to both the processor and the monitoring hardware. The ISR monitoring graph is always resident in monitoring graph memory. If the monitor detects an illegal instruction execution during the execution of the ISR, it resets the processor. The processor can elect to disable interrupts. In this case, the processor writes to a register in the monitor indicating that it should ignore future IRQ strobes. The monitor keeps tracking the processor until the processor writes the *disable interrupt* command into its status register.

• System Calls: Another source of context switching is *system calls*. During these calls, the user function is suspended until the operating system returns control back to it. For monitoring, system calls are assigned a GID and control is passed to a monitoring graph for the system function. When a user task calls a system function, the GID of the function is determined from a field embedded in the user task monitoring graph for the executed instruction. Once the GID for the system call is determined, the hardware monitor saves the state of the current task's monitoring in

the bookkeeping tables, finds the memory slot holding that system call's graph using its GID and starts traversing that graph. During the automatic loading of the system call's monitoring graph, the CPU is stalled by the deactivation of the *Done* signal from the monitor.

• OS Scheduler: The most common source of context switches is the OS user task scheduler. The two most frequent invocations of the OS scheduler are from the timer ISR and an application's system call to yield the processor. When the scheduler is invoked, it chooses the next process to execute. In μC/OS-II, a priority-based scheduler is used, although round-robin or other approaches are also possible. When the next task is determined, the processor forwards the task's PID to the monitor. The monitor identifies the appropriate monitoring graph by consulting the bookkeeping tables. Once the monitoring graph information is in place, the context switch is made and monitoring is switched from the ISR or system call monitoring graph to the graph for the user task. The processor is notified that it can proceed via the *Done* output from the monitor.

3.1.3.4 Recovery

When an attack is detected, the monitor signals a reset to the processor. For application processes, the operating system can simply kill the process and use internal mechanisms to recover memory and restart the application. Many embedded applications can recover from such a restart. If necessary, more complex checkpointing and recovery mechanisms can be implemented. If the reset occurs during operating system processing, then the entire system needs to be restarted.

One concern with the recovery process is that a simple attack (e.g., caused by a small number of I/O operations) can cause a costly recovery operation (e.g., rebooting the system). An attacker could use this as a denial-of-service mechanism. However, the hardware monitoring system ensures that the attack does not succeed (i.e., no

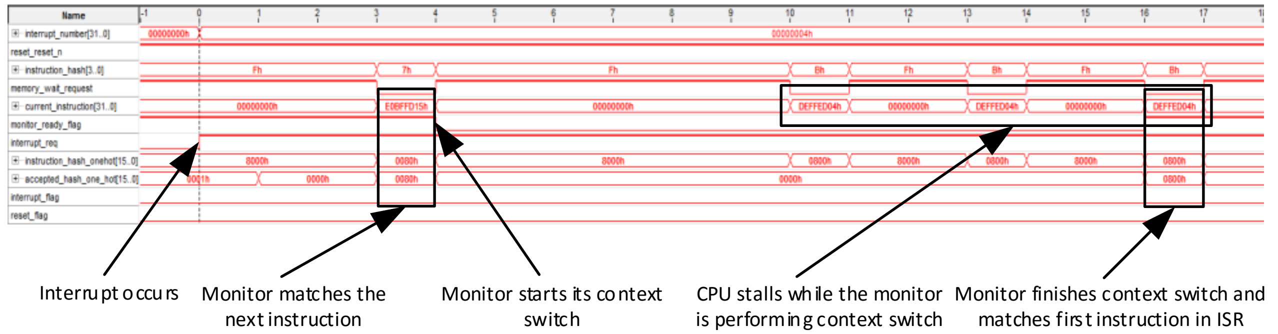


Figure 3.3. Context switch interactions between processor and monitor.

Figure 3.4. Attack on processor system, which is detected by hardware monitor.

malicious code is executed) and no attack vectors beyond this denial of service are available to an attacker.

3.1.4 Prototype Implementation

We have developed a prototype implementation of the hardware monitoring system to show its effectiveness in providing security to embedded operating systems and their applications.

3.1.4.1 System Setup and Attack Scenario

Our system consists of a simple NIOS II soft processor, which is augmented by our hardware monitor. The system is described in Verilog and implemented on Terasic DE4 FPGA board utilizing an Altera Stratix IV FPGA. To install new binaries and

graphs, another NIOS II core with a dedicated RSA decryption engine for secure hardware monitor loading is co-located with the main processor and the hardware monitor. The role of this co-located processor is to read encrypted binaries and graphs from an SD card, decrypt and verify them, and feed them to the main processor and the hardware monitor. Security installation of the binaries and graphs are discussed in detail in [34].

To test the ability of the hardware monitor to detect runtime attacks, we implemented a format string attack scenario [18]. Exploiting the vulnerability in `snprintf` function, we overwrite the first instruction of the interrupt service routine and replace it with a call to an arbitrary function which prints a simple message on the console. In a practical attack, this redirection of control flow can be used to execute arbitrary attack code.

3.1.4.2 System Operation

Under normal operation, our hardware monitor follows along with the context switches that occur in the operating system. When switching from an application process to the operating system (or the other direction), the current monitoring context is stored and the new monitoring context is loaded.

Fig. 3.3 shows the interactions that take place during such a context switch. Our hardware monitor is blocking in the sense that the embedded processor stalls if the processor switches between contexts more quickly than the monitor (see cycles 10–16 in Fig. 3.3). While this stalling causes a slight overhead (see Section 3.1.4.5), it ensures that no instruction is executed without being monitored.

3.1.4.3 Detection of Attack

When the processor is being attacked and the execution of attack code is attempted, the monitor reports an instruction execution that does not match with the monitoring graph of the application binary or operating system that is currently ac-

Table 3.2. Monitoring graph sizes for operating system and applications.

	number of instructions	number of graph entries	graph size (bits)
μC/OS-II	22,913	23,625	850,500
basic_math	10,446	11,563	416,268
bitcount	6,731	7,823	281,628
qsort_small	7,113	9,055	325,980
qsort_large	7,302	9,116	328,176

tive. Fig. 3.4 shows such an attack detection. In particular, at cycle 18, there is a difference between the reported hash value (0x0008, i.e., 3 in one-hot coding) and the acceptable hash values (0x0800, i.e., 11 in one-hot coding). Thus the reset signal is asserted.

These results show that the security requirements (SR1 and SR2 in Section 3.1.2.3) are met. In particular, as long as the attacker cannot modify the monitoring graphs (AC3), any change in the processing system (AC1 and AC2) that leads to any change in processing behavior can be detected by the monitoring system.

3.1.4.4 Monitoring Graph for Benchmarks and the OS

Using the graph extraction method described in [13], we extracted the monitoring graph for μC/OS-II and a set of benchmarks from [28]. To run the benchmarks on our NIOS based platform, minor modifications were performed. For example, our system does not have a file system. Therefore, we had to use static predefined data sets instead of reading from files. The number of instructions, the number of graph memory entries, and the total graph sizes in graph memory for each benchmark and the OS are shown in Table 3.2.

3.1.4.5 Monitoring System Overhead

There are two types of overhead that we need to consider for our system. One overhead is the additional on-chip area that is required by the hardware monitor. This area consists of the logic necessary to implement monitoring functionality and

Table 3.3. Resource use on a Stratix IV FPGA.

	Available on FPGA	Nios II w/o HW mon.	HW monitor + controller	Secure HW mon. loading
LUTs	182,400	2,997	764	2,603
FFs	182,400	3,200	922	2,936
Mem. bits	14,625,792	2,199,552	2,580,288	977,332

context switching and the memory that is necessary to store monitoring graphs and contexts. We show the resources necessary for implementing the hardware monitor in Table 3.3. The hardware monitor requires less logic than the embedded processor since its functionality is much simpler. It does require comparable memory resources because the monitoring graph needs to be stored in a format that allows fast transition between states (within one processor cycle). In addition, the system requires a mechanism for securely loading monitoring graphs (to avoid tampering by an attacker). This security mechanism requires resources comparable to that of the processor and is shown in the final column of the table.

From these resource figures, the hardware monitoring system may seem *relatively* expensive to implement. However, the *absolute* resource use is very small. Also, the cost of the monitor does not increase with a higher-performance processor. For example, a higher-end processor may require more logic and have significantly more data memory, but the monitoring system would require the same amount of resources. Also, the secure loading system would only be required once when using a multi-core embedded system. Thus, the overall resource consumption is practically feasible.

The other overhead is the processing delay that is introduced by stalling the processor core during a context switch (see Section 3.1.4.2). The delay for an interrupt on the processor (without any monitoring in place) is 6 cycles. As Fig. 3.3 shows, an additional 6 cycles of stalling is introduced by the hardware monitor which is not comparable to the hundreds of cycles needed to execute the ISR itself. The effect of this additional delay depends on the frequency of interrupts in the system. It should

be noted that the original NIOS based system had the highest possible clocking rate of 198MHz and adding the hardware monitor and the cryptographic processor to it did not impose any slow down in terms of maximum clocking frequency.

These results show that our hardware monitoring system, which can detect any attack that changes processing behavior, can be implemented with reasonable amounts of additional hardware resources and practically no performance degradation on the system.

3.2 Hardware Monitoring in Mid-Range IoT Devices

In this chapter, so far, we described a hardware monitoring approach to protect embedded systems running a lightweight embedded operating system. Running an operating system enables such embedded systems to run multiple tasks at the same time. The main contribution of the work presented in this chapter was to extend the hardware monitoring solution proposed in Chapter 2 to seamlessly follow the execution of the code (application of OS) running on the embedded processor throughout the entire operation of the system.

The embedded system we described here was a good representative of low-end embedded systems. However, there are other types of embedded systems already in use whose characteristics do not quite match the representative system we discussed here. The main difference between such embedded systems and our represented system mostly relies on the amount of processing power they have. Having more resources enables embedded systems to run more complicated operating systems which consequently provides a better platform for a variety of embedded applications. One of the most prominent classes operating systems in the jargon of embedded systems is distributions of Linux kernel. We explore the challenges of monitoring an embedded system running Linux in a separate work [58]. The main barrier we faced in that work was the big size of the Linux codebase which results in a big monitoring graph for the

kernel. Using the onchip memory available on the FPGA for storing the monitoring graph, we could not afford such a high memory requirement. One compromise we made in that work was to limit the scope of the monitoring to the system calls as the first line of defense in the OS.

3.3 Summary and Conclusions

In summary, embedded systems are particularly vulnerable to attacks. Their limited resources do not allow the use of conventional software-based defense mechanisms. In this chapter, we presented a hardware-based security mechanism that can ensure correct execution of applications *and* operating system code at the granularity of individual instructions. Our prototype system shows that the proposed mechanism is feasible for these highly dynamic environments and effective in detecting any attack, even those that were previously unknown. We believe that this work presents an important step towards secure embedded processing systems for a broad range of applications domains.

CHAPTER 4

A LIGHTWEIGHT PAYMENT VERIFICATION PROTOCOL FOR BLOCKCHAIN TRANSACTIONS ON IOT DEVICES

Over the past two decades, the idea of Cyber-Physical Systems (CPS) and their use in a broader network, known as the Internet of Things (IoT), has attracted a lot of attention as the technological foundation to address many important societal problems relating to the environment, healthcare, transportation, etc. [27, 60, 65, 67, 80]. Fundamentally, IoT systems interconnect three types of components: *sensors* that detect or measure a physical property, *computation* that receives sensor data for processing and making control decisions, and *actuators* that act in the physical world in response. These three steps of sensing, computing, and acting can vary vastly in their implementation based on the IoT application and use case. However, the basic control loop that spans the physical and computational world can be found throughout the IoT space.

Conventionally, each IoT application needs to deploy its own set of sensors and actuators in the environment, provide network connectivity to them, and implement its computation/control algorithm on a dedicated computing system. The capital cost of setting up such *vertically integrated* IoT solutions can be a significant impediment to the adoption and wide-spread use of IoT-based solutions. In order to achieve large-scale use of the Internet of Things, we have argued that it is equally important to enable a *horizontally integrated* Internet of Things [78].

Horizontal integration allows for IoT components that belong to different administrative entities to be combined together dynamically to implement a new IoT appli-

cation. For example, an autonomous drone could use a local weather sensor to query wind conditions to determine its flight path. Clearly, deploying wind sensors along all the potential paths of such a drone would be cost-prohibitive. However, reusing a weather sensor, which has already been deployed for local weather forecasting or an agricultural monitoring application facilitates this new IoT application without the need for a new sensor deployment. In the IoT industry, we see the first steps towards such a common IoT infrastructure on the computation side. Several vendors, such as Amazon IoT as part of Amazon Web Services [2], Microsoft Azure [44], and Google Cloud IoT [26] offer cloud-based IoT platforms that remove the need for dedicated computing infrastructure. A logical next step is to develop environments that also allow sharing of sensors and actuators.

A key question in horizontally integrated IoT systems is why anyone would want to share access to their sensors or actuators. Some systems, such as the early Internet, rely on all users to cooperate for the common good. Such a premise is difficult to maintain in IoT environments, where sensors and actuators may have significant costs of deployment and operation. Thus, it is critical to develop a basis for *economic transactions* in IoT. Access to sensors, actuators, and computation can be offered as services that IoT users and their applications can access in return for payment.

As we monetize the data collected by these devices and get financial transactions involved in the context, these IoT devices become attractive targets for attackers. In the previous parts of this work, we focused on how to strengthen the hardware platform of these IoT devices against remote attacks. While a hardware monitor can make sure that the running software is sticking to its original intended behavior, it is of no use if the software is not protecting the assets it controls properly and leaks information which could jeopardize the privacy of those assets.

Clearly, there are many interesting technical questions on how to build a fully interoperable, horizontally integrated IoT environment. For example: How can we

build an IoT marketplace so that applications find suitable services [79]? How can we achieve interoperability between components? How can we manage resources on actuators? How can we achieve privacy and security? We have alluded to some of these challenges in prior work [56, 78] and by no means do we or the community have answers to all of them yet. In this chapter, we put some of these challenges aside and focus specifically on the question of how to implement economic transactions between IoT components that belong to different administrative entities. In particular, we consider the challenge that many typical IoT components are based on low-end embedded systems with limited computational and communication resources. Thus, we aim to find solutions that limit computational demands and that reduce communication through a decentralized design.

We present a Ticket-Based Verification Protocol (TBVP), a technique for enabling economic transactions in IoT systems that can meet practical constraints. TBVP uses blockchain [49] technology as the foundation for establishing an economic value for IoT transactions. However, running a blockchain client requires high processing capability as well as high network bandwidth, demands that are not always available to IoT devices. By introducing two logically separated entities, a *contract manager* and a *transaction verifier*, in TBVP, we reduce the demand for running a blockchain client to only one entity per administrative domain. As a result, we reduce the processing requirement of the IoT devices to simple encryption/decryption and signature verification. By doing so, we enable any IoT device, regardless of processing power and available network bandwidth, to horizontally integrate in an IoT economic ecosystem.

The specific contributions of our work are:

- Design of a Ticket-Based Verification Protocol that enables performance-constrained IoT devices to participate in financial transaction in an IoT ecosystem.

- Implementation of a prototype system that realizes blockchain-based transactions on typical low-end embedded systems.

- Presentation of results showing that the proposed approach is effective in performing payment transactions with low performance requirements.

Our work shows that our TBVP system is an effective approach to enabling economic transactions in an IoT system. The economic relationships that can be established using such transactions enable interactions between IoT components from different administrative entities and thus enable horizontal integration in future IoT systems.

The remainder of this chapter is organized as follows. Section 4.1 briefly discusses related work. The connection between horizontal integration and the need for economic transactions is argued in Section 4.2. Section 4.3 describes the current state-of-the-art blockchain solutions and their drawbacks. The details of our Ticket-Based Verification Protocol and how it is able to address those drawbacks are discussed in Section 4.4. A prototype implementation and its results are presented in Section 4.5. Section 4.6 summarizes and concludes this chapter.

4.1 Related Work

An economic transaction encompasses several aspects: typically, there is a commodity or service that is provided and a payment that is given in return. In the context of information technology systems, these transactions are sometimes referred to as "smart contracts" [71]. The idea of paying for access to IoT components was described in our work [79] and later by Worner et al. [82]. The confluence of embedded system technology, cryptocurrencies, and smart contracts is described by Omohundro [52]. A brief survey by Conoscenti et al. describes some work in this area [17]. The work by Zhang et al. [85] describes an e-business architecture that focuses on smart contracts in IoT systems. The concepts of blockchains are referenced in this context, but no clear use or deployment ideas are provided.

The work by Christidis et al. [16] describes the use of blockchains to establish trust in an IoT system to facilitate sharing of services and smart contracts. Similarly, Bahga et al. [7] use blockchains in an Industrial Internet of Things (IIoT) context to achieve trust among components. Neither work uses blockchains in an economic sense, i.e., there is no explicit exchange of payment between components. The work by Huckle et al. [35] describes how the Internet of Things can use blockchains to enable applications for the sharing economy. This work has a similar goal as ours, i.e., enabling economic rewards for all players in an ecosystem, but proposes to use IoT components to enable payments (e.g., near-field communication to authorize credit card payment). We aim to enable economic exchanges between IoT components for use of IoT services.

The idea of using offchain transactions to reduce the overhead of these transactions is described by Heilman et al. [30]. The Raiden Network described in [29] uses offchain transactions to reduce transaction overhead. This work describes a simple transaction scenario for IoT systems, however its processing power requirement and network traffic consumption make it almost impossible for such solutions to be adopted in IoT setups where low-end devices are used. Our work describes a protocol that enables IoT devices within different administrative domains to exchange cryptocurrencies and data regardless of their processing capability and available network bandwidth by introducing a logically centralized gateway within each administrative domain.

4.2 Horizontally Integrated IoT

Before presenting the details of our TBVP architecture in Section 4.4, we briefly discuss the value of horizontal integration in IoT and the economic relationships that are formed among IoT components.

4.2.1 The Need for Horizontal Integration

Deployment of IoT solutions can be expensive since sensors, actuators, and computation components are necessary. Traditionally, each application uses its own components. Switching from dedicated computation infrastructure to cloud-based computation platforms can reduce some of this cost. However, each application still needs to set up and maintain its own set of sensors and actuators in the environment. An alternative approach, which can make this investment more cost effective, is to share that platform with other IoT applications. In order to facilitate that resource sharing, we proposed *horizontal integration* of IoT components [78, 79].

4.2.2 Economic Relationships and Systems

Having well-defined architectural guidelines to ensure interoperability is the first step in adding support for horizontal integration. However, given the cost of deploying and operating sensors and actuators, it is not very likely for IoT infrastructure owners to voluntarily share their resources with others. However, it can make economic sense for the owners to do so if they can be compensated for the service they provide. We believe it is crucial to have a proper economic relationship between service providers and consumers in order to incentivize platform owners to share their resources.

Given the nature of IoT systems, the amount of money being transferred between IoT systems is typically small, but the frequency and diversity of those transactions may be high. This is exactly the opposite of what current financial mechanisms are good at. The current financial systems rely on a logically centralized ledger that keeps track of account balances and validates and commits transfers between them. The main drawback of such systems is the cost of setting up and running that centralized entity. The cost of having such a capable centralized ledger imposes high fees on each transaction it processes.

An alternative to the current centralized ledger approach is the use of a blockchain [49], which decentralizes and democratizes the financial system by letting the parties themselves take care of the bookkeeping and validating of transactions. By doing so, blockchains eliminate the need for a centralized ledger and therefore remove the capital cost of having such a central entity. As a result, blockchains impose lower overhead or fees for each transaction.

In terms of blockchain networks, there are multiple systems that have been developed, such as Bitcoin [9], Ethereum [20], and Litecoin [40]. We chose to base our system on Ethereum since it offers a powerful protocol for building applications in a decentralized manner. Those applications include, but are not limited to, financial applications, which are described in more detail in [11]. Ethereum's cryptocurrency, Ether, is also one of the most popular cryptocurrencies in the market, which means it is more likely for our system to be adopted as a solution in practice.

4.3 Current Blockchain-Based Solution

In this section, we first discuss the basics of current blockchain-based networks and how they operate and address the challenge of high transaction fees and low number of transactions per second associated with main-chain-based transactions. Then, we discuss the typical drawbacks of such approaches and how our proposed solution addresses those challenges.

4.3.1 Operation

As discussed above, a large number of small transactions are necessary for an IoT system. One efficient way to use a blockchain for a large number of small transactions is the use of *payment channels* [10]. A payment channel is useful when two parties are doing business with each other frequently. The idea is for both parties to keep

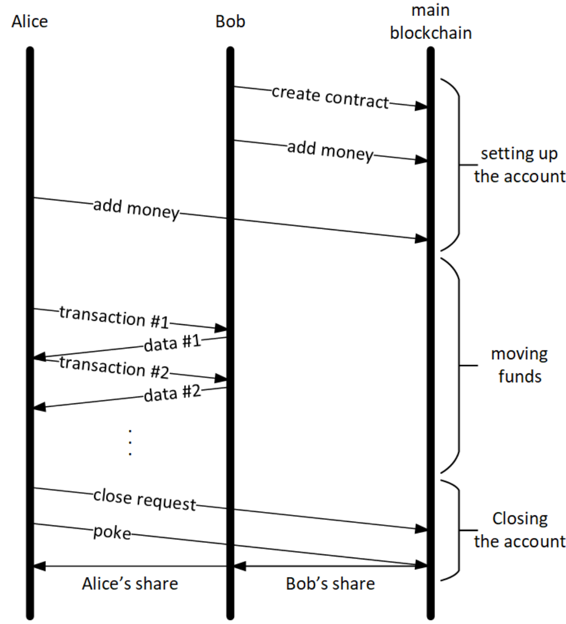


Figure 4.1. Space-Time diagram for money and data exchange through payment channel as proposed by Poon et al. [54].

track of fund transfers between themselves (in a provable format) and then settle the balance only once on the blockchain when they are done.

Payment channels can be implemented using a smart contract in Ethereum somewhat similar to joint accounts in conventional banking system but with a slight difference. The parties open a joint account and each of them initially puts some amount of money in that. Then, when they do business, instead of transferring actual money, they transfer ownership of their shares from the funds locked up in that bank account. In order to avoid paying bank fees, these ownership transfers should be done offline but in a provable way. Finally, after they are done with their transactions, the parties go to the bank with the most up-to-date version of their balance sheet and split the money accordingly. Although this process seems simple at the first glance, there are some nuanced implementation considerations to make sure nobody is able to cheat and both parties are able to withdraw their money from the joint account at any time they want. The space-time diagram for these interactions is shown in

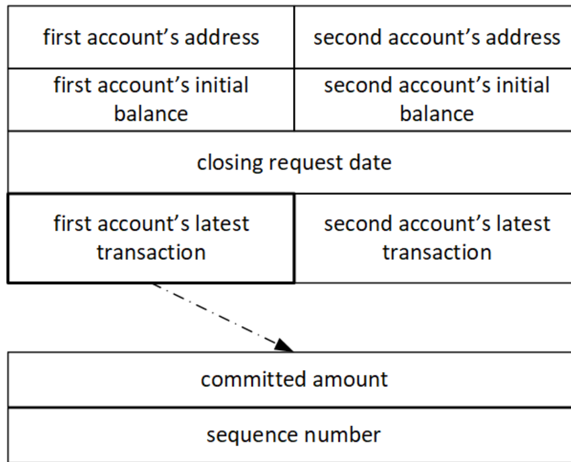


Figure 4.2. Data structure used in the smart contract.

Figure 4.1. Imagine Bob and Alice are two business partners who have a lot of business transactions on a regular basis. They are also willing to use payment channels in order to avoid high transaction fees associated with regular transactions on the main chain. The following describes the steps they need to take in order to setup and use an off-chain payment channel.

4.3.1.1 Setting Up The Joint Account

Alice first provides Bob with her account address so Bob can go ahead and create a smart contract (the joint account) on the blockchain. After creating the smart contract, Bob moves some amount of money to that smart contract. This new contract will be an account by itself, which has an address. By providing that address to Alice, she can read the code being run on that contract and the amount of money Bob has moved to that contract. In the contract, there is a variable called *commited_amount* that determines how much of the money being held by the contract is promised to Alice. The contract should provide a mechanism for both Bob and Alice to call off the arrangement and withdraw their share anytime they want regardless of what the other party wills. It should also provide a mean to change the value of the *commited_amount* variable determining Alice's share.

4.3.1.2 Moving Funds

In order to transfer money, Bob only needs to change the amount of money he promised to Alice by changing the *commited_amount* variable in the contract. But sending messages to the contract has to be done through the main blockchain and therefore imposes transaction fees. To avoid these fees, instead of updating the value of that variable on the main chain, every time he wants to send a payment, Bob crafts a message which updates that variable [55]. We call that message *commit message* here. Bob then signs that commit message and sends it to Alice (instead of sending it to the contract). Having that message, Alice is assured that she can claim her money anytime by just publishing that message, destined to the contract address, on the blockchain. In our example, it is only Bob who pays Alice, which makes the payment channel unidirectional. In case a bidirectional payment channel is needed, an extended version of the contract can be used, where there are two variables, one for each party's commitment to the other.

4.3.1.3 Closing the Account

One major difference between the proposed Ethereum smart contract and the joint accounts in the conventional banking system is that in the conventional banking system each withdrawal request should be signed by both parties. This requirement enables each party to blackmail the other by not agreeing to sign a withdrawal transaction when the other is willing to. We avoid this problem by letting each party request to close the joint account by sending a *close request* message to the contract. Prior to that message, the party willing to close the account should have submitted the most recent commit message signed by the other party to update the value of the *commited_amount* variable. When one party requests to close the account, the account cannot close immediately since the other party might not have published their latest commitment messages yet. Therefore, after receiving the close request,

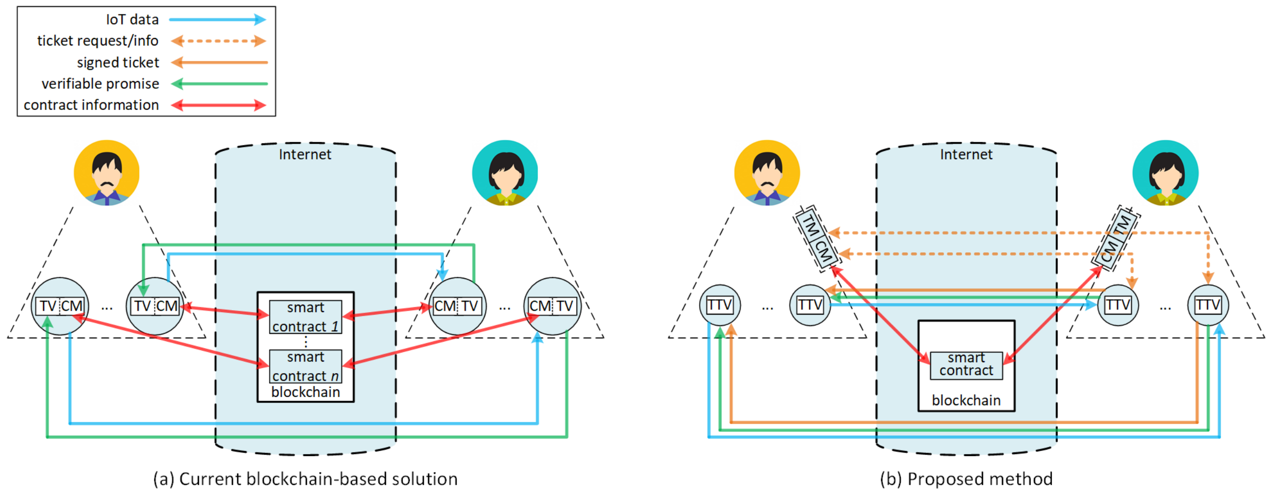


Figure 4.3. Flow of information and money in (a) current blockchain-based solutions versus (b) our proposed Ticket-Based Verification Protocol

.

the contract has to wait for a period of time called *complaint free* period. During this period, each party has a chance to update their submitted commit message. This complaint free period will be restarted after a party submits a new commit message. In order to avoid stalling the contract forever by either parties, we add a sequence number to each commit message. The contract accepts a commit message only if its sequence number is bigger than the last message received. Figure 4.2 shows the data structure used in the smart contract between the parties.

4.3.2 Drawbacks

So far, we have briefly discussed how the state-of-the-art blockchain-based networks utilize payment channels to facilitate fast and low-fee transactions between business partners. Figure 4.3(a) demonstrates how information and money flow through the blockchain, as well as off the blockchain, in a conventional payment channel setup. In such setup, every single participating IoT device (both buyers and sellers) needs to be connected to the blockchain in order to:

1. Set up a smart contract for every business partnership;

2. Constantly monitor the blockchain for changes to the associated smart contracts;

3. Receive and validate transactions targeting the smart contracts when providing a service to another node; and

4. Send a valid transaction to spend from a smart contract in order to purchase a service or information when buying.

Requiring all the participating devices to be connected to the blockchain network implies a running blockchain client on those devices. However, running a blockchain client imposes high incoming network traffic to the node and demands high processing power in the device. Having access to high network bandwidth is not always the case for IoT devices and most such devices do not utilize high-end processing units. Another impediment to widespread use of current blockchain-based approaches is the need for separate blockchain accounts for each of the devices participating in horizontal integration. Having multiple accounts means setting up multiple escrow services and locking up a bigger sum of money for a single administrative domain. Although internal transactions within payment channels are done off the chain and do not incur fees, setting up payment channels and closing them should be done through the main chain and therefore there are fees associated with such operations. We can summarize the drawbacks of current blockchain-based solutions as follows:

- High communication and processing overhead imposed on IoT devices.
- Strong reliance on the operating system and therefore the lack of portability.
- Requirements for multiple blockchain accounts and therefore high cost of consolidating the accounts.

Our proposed Ticket-Based Verification Protocol is designed to address these three major challenges and therefore paves the way of using blockchain-based transaction

mechanisms on widely used low-end IoT devices. In the next section, we discuss the details of our TBVP and describe how it addresses each of these issues.

4.4 The Ticket-Based Verification Protocol

In Section 4.3, we presented existing solutions and summarized the three major drawbacks of such approaches. In this section, we discuss the details of our Ticket-Based Verification Protocol and describe how it addresses those three challenges.

4.4.1 Blockchain-Based and Blockchain-Agnostic Operations

The tasks performed by IoT devices to implement financial transactions can be divided into two categories: *blockchain-based* and *blockchain-agnostic* operations. *Blockchain-based* tasks include setting up a smart contract, moving money into the contract, sending commit messages (verified promises) to the contract, and closing the contract and claiming the funds. In contrast, receiving a commit message from a partner and validating that message is something that does not need any connection to the blockchain and therefore is a *blockchain-agnostic* operation. This logical division is shown in Figure 4.3(a), where each IoT node has a Transaction Verifier (TV) taking care of *blockchain-agnostic* operations along with a Contract Manager (CM) which is responsible for handling *blockchain-based* operations.

4.4.2 Separation of Contract Manager and Transaction Verifier

The IoT node uses the CM unit initially to set up a joint account with a partner device and deposit funds into that account. After the account has been set up, the device is ready to provide units of service (it could be sharing data or acting upon a remote command) in return of provable promises:

- Buyer of service: In order to receive a unit of service, the TV of the buyer node forms a promise consisting of the cumulative amount of money for the

past services and the unit being bought. The TV then increments the sequence number from the last promise, packs that with the cumulative amount, and signs the package using the secret key initially provided by the CM. The signed package is then sent to the service provider node to pay for the service being bought.

- Seller of service: A provable promise is received from the sender. Upon receiving a promise, the TV checks if the amount of money is what was promised and if the sequence number has been incremented from the last received promise. If the promise passes those two checks, the service can be provided to the partner device in return.

The above steps can be repeated many times. When the two IoT devices are done with data and money exchange, on the provider side, the TV passes the latest promise to the CM. The CM then submits that promise to the smart contract to claim its money. On the buyer side however, there is nothing that the CM should do unless it does not want to wait for the provider's CM to close the contract. In this case, the CM of the buyer can go ahead and submit a request to the contract to close it. In this case, the buyer's CM, which is watching the contract on the blockchain, will have to submit the latest promise it has received in order to claim its share of money.

Tasks performed by the CM are inherently heavy in terms of processing and network traffic consumption as they require constant interaction with the blockchain. One important observation about these tasks, however, is that they are only used rarely, which is the key design point of the TBVP. In the TBVP, we define two separate entities: *Gateway* and *Things*. The main responsibility of the *Gateway* is to run an instance of CM, whereas *Things* only need to run a TV instance. In order to participate in horizontal integration, every administrative domain only needs to have one logically centralized *Gateway* to support multiple IoT devices acting as *Things*. Since CM and TV are not co-located on the same device, local communica-

	ParentTxHash	Block	Age	From		To	Value
7:	0x2e8216306601909...	2253162	57 secs ago	0xa335e4ddbedf656...	→	0xc8cccea55a9bb54...	6 Ether
	0x2e8216306601909...	2253162	57 secs ago	0xa335e4ddbedf656...	→	0xf45449ca9a1b56f...	4 Ether

	TxHash	Block	Age	From		To	Value	[TxFee]
6:	0x2e8216306601909...	2253162	57 secs ago	0xc8cccea55a9bb54...	IN	0xa335e4ddbedf656...	0 Ether	0.000036178
5:	0xa0c933669a1000a...	2253154	3 mins ago	0xc8cccea55a9bb54...	IN	0xa335e4ddbedf656...	0 Ether	0.000042064
4:	0xeeb4923ecb5520b...	2253130	8 mins ago	0xc8cccea55a9bb54...	IN	0xa335e4ddbedf656...	0 Ether	0.00042064
3:	0x9067cb5aa493585...	2253110	13 mins ago	0xf45449ca9a1b56f...	IN	0xa335e4ddbedf656...	5 Ether	0.000188151
2:	0x561484ecd124aa5...	2253108	14 mins ago	0xc8cccea55a9bb54...	IN	0xa335e4ddbedf656...	5 Ether	0.000042389
1:	0x8acc4b4540be58e...	2253100	16 mins ago	0xc8cccea55a9bb54...	IN	Contract Creation	0 Ether	0.007473345

Figure 4.4. Blockchain transaction details for experiment 1.

tion mechanisms are used to coordinate. In our approach, communication between these entities is done using *Tickets*. Figure 4.3(b) shows the separation of CM and TV in our approach. Ticket Manager (TM) is a new functionality added to the gateway in addition to CM. On *Things*, Transaction Verified has been extended to Transaction/Ticket Verifier (TTV) to verify the tickets issued by the TM.

4.4.3 Transactions using Tickets in TBVP

Tickets in TBVP are very similar to those in Kerberos [51]. In TBVP, tickets are used to authenticate legitimate service buyers in a horizontally integrated IoT setup. Let us assume that there are two IoT devices, i_1 and i_2, within two different administrative domains, A_1 and A_2, respectively. When i_1 wants to access a service provided by i_2, it has to send a ticket request for that service to A_1's gateway, g_1. Receiving that service request, g_1 tries to establish a payment channel (or in more complex scenarios, a path of payment channels) between itself and A_2's gateway, g_2. After having a payment channel formed between g_1 and g_2, the gateways agree on the price of each unit of service being requested.

Next, g_1 notifies the g_2 about the maximum number of service units needed. Upon receiving this request, g_2 makes sure that there is enough money available in the joint account for that number of service units at the negotiated price. Gateway g_2 then is-

sues a ticket, which consists of the following fields: *buyer's address*, *buyer's blockchain account*, *i_2's ID* (assigned uniquely within A_2), *expiration date*, *total number of service units* i_2 is allowed to provide, *price per service unit*, and *starting sequence number.*

The ticket is then signed with the secret key of g_2 and sent back to g_1. Upon receiving the ticket, g_1 passes the ticket to i_1. In addition to the ticket, i_1 also needs the secret key of the blockchain account through which the promised money should be spent. From this point on, i_1 and i_2 can directly communicate with each other without going through the gateways. In order to initiate a connection, i_1 sends the ticket to i_2 whose service is desired. Upon receiving the ticket, i_2 recognizes i_1 as a legitimate client and can serve it in return for verifiable promises spending from the account address set in the ticket.

4.5 Implementation and Evaluation

In this section, we describe the implementation of a prototype system implementing our Ticket-Based Verification Protocol to exchange IoT data in return for money. Our results show that the system works effectively and efficiently, even when using low-performance embedded system, such as those commonly used in IoT.

4.5.1 Experiment 1: Sharing Sensor Data

In order to show the applicability of our design approach to a real world scenario, we implemented two IoT administrative domains, called A_1 and A_2 here. Each of these administrative domains has a Raspberry Pi 3 acting as *Gateway*, and a TI CC3200SF LaunchPad Kit acting as *Thing*. We refer to the gateway and the thing of each administrative domains as g_x and i_x respectively where the index represents the corresponding administrative domains.

In this scenario, i_1 and i_2, each reads its temperature sensor. i_1 is programmed to buy i_2's data and compare it against its own reading, for 1,000 samples. Whenever

the difference between those values is greater than 1° F, i_1 turns one of its LEDs on alarming the discrepancy.

In this experiment, there is a payment channel formed between A_1 and A_2, into which, each of them initially deposits 5 Ether. The price for each piece of temperature data is set to 0.001 Ether which means that A_1 should eventually pay 1 Ether to A_2 for 1,000 units of data.

Figure 4.4 shows the details of transactions on the blockchain. A_2 (address 0xc8cc...) initially creates a smart contract and deposits 5 Ethers into it (steps 1 and 2). A_1 (address 0xf454...) then joins the contract by depositing 5 Ethers into it (step 3). After that, all the communications are done off the blockchain. When done with the 1,000 transactions, A_2 submits the last promise it has received from A_1 to the blockchain (step 4) and requests to close the contract (step 5). A_2 then triggers the contract to close (step 6). Finally, the contract terminates and distributes its 10 Ethers (step 7) according to the commit message it has received in step 4 and sends 6 Ethers to A_2 and 4 Ethers to A_1.

4.5.2 Experiment 2: Performance Evaluation

After ensuring the correct behavior of our approach, we evaluated the performance of the Raspberry Pi 3 and the CC3220SF acting as an IoT device in two different setups: (a) μRaiden [74] (i.e., the traditional block-chain-based approach described in Section 4.3) and (b) our new TBVP. In order to isolate the performance of each device under test from the outside impacts, each device has been tested in a separate experimental setup where all other functionalities within the network have been running on a desktop computer with Intel Core i7 4770K CPU and 16 GB of RAM. The network connectivity between the desktop computer and the device under test was through a local network switch with the desktop computer connected to the switch through Ethernet and the device under test through WiFi.

Table 4.1. Number of Operations per Second Performed by Each Device in Different Roles in μRaiden versus TBVP.

	Serving Thing			Requesting Thing		
	μRaiden	TBVP	speedup	μRaiden	TBVP	speedup
PI 3	7	43	6.14x	5	65	13.0x
CC3220	unavail.	3	new	unavail.	4	new

Table 4.1 shows the number of transactions per second performed by different devices (contract creation time not considered). As summarized in the Table, the Raspberry Pi is able to serve 43 requests per seconds in TBVP while this number is 7 while implementing μRaiden which shows 6.14x better performance by TBVP over μRaiden. On the requesting side however, Raspberry Pi 3 is able to form and send 65 request packets per second while implementing TBVP. This number is 5 for the same device performing similar functionality while implementing μRaiden, which shows 13.0x improvement in performance by TBVP over μRaiden.

Finally, Table 4.1 shows 3 and 4 operations per second for CC3220SF for serving requests and generating requests while implementing TBVP. On μRaiden, none of these operations can be implemented on the low-end CC3220SF device. Thus, these results show that our TBVP enables a new class of devices to engages in economic transactions in IoT.

4.6 Summary and Conclusion

The cost of deployment and operation of IoT devices is the main impediment to widespread use of IoT-based solutions in different societal problems. Horizontal integration of IoT devices is a promising approach to overcoming the cost barrier of such solutions. Having a proper financial mechanism is the crucial first step in the horizontal integration of IoT devices. In this chapter, we proposed a Ticket-Based Verification Protocol which is an extension of the current blockchain-based financial mechanisms for IoT devices. Unlike the current blockchain-based mechanisms,

it does not require all the IoT devices to participate in complex interactions with the blockchain itself and therefore this protocol can be used in various IoT setups, including those with low-end IoT devices.

CHAPTER 5

SECURING THE EXCHANGE OF DATA AND MONETARY VALUES ON MODERN IOT DEVICES: A MULTI-LAYER APPROACH

Embedded systems are at the core of the Internet of Things. As Moore's law progresses, these embedded systems have moved from simple microcontrollers to full-scale embedded computing systems with multiple processor cores and operating systems support. Given the fast-paced nature of the IoT industry, the software applications for these devices are mostly developed with one major goal in mind: delivering the promised functionality in the shortest possible time. This mindset results in the security of such applications being usually overlooked.

Recent attacks carried out by IoT devices including the DDoS attack by a Mirai botnet [3] exploiting CCTV cameras show how vulnerable IoT devices can be. On the other hand, the large-scale deployment of such IoT devices all over the world makes the collective power of such botnets very destructive [24, 38]. Security of such devices, like any other network-connected devices, can be discussed on two different levels: *protocol* and *execution integrity*.

In Chapter 4 we described a secure protocol that can be used by a variety of IoT systems to securely exchange data for cryptocurrencies. However, the use of this protocol, like any other protocol, can be effective only if the IoT devices follow the steps dictated by the protocol. Runtime attacks which cause the embedded processor to deviate from its intended behavior, are one of the main threats to proper execution of secure protocols.

In the first two chapters of this work, we described how the current hardware monitoring solutions are based on the assumption of having an embedded system with bare-metal execution of a single application. We further explained our approach to expand the capabilities of hardware monitors to consider multi-application workloads as well as the presence of an operating system.

Despite this progress in hardware monitor design, the most general case of a multi-application workload on a multi-core system that is controlled and scheduled by an operating system has not been successfully tackled. This scenario is particularly challenging due to the dynamism that is introduced by arbitrary starting and stopping of processes, interrupts, and scheduling and migration of processes across cores.

In this chapter, we address the challenges of adding a hardware monitoring system to modern IoT devices running a full-scale Linux kernel on a multi-core processor. We also show how the Ticket-Based Verification Protocol we proposed in Chapter 5, can still be vulnerable against runtime attacks despite being secure at the protocol level.

The rest of this chapter is organized as follows: In Section 5.2, we described the implementation of the TBVP protocol on a LEON3 processor. We also evaluate the performance and correctness of the protocol in this section. In Section 5.3, we initially described how this protocol can be implemented in a way that is vulnerable against runtime attacks and how those vulnerabilities can be exploited to hijack the system and cause financial losses. In the second part of the chapter, we describe our hardware monitor solution to protect this implemenation against those runtime attacks.

In Section 5.4 we extend our hardware platform to utilize a multi-core implementation of LEON3 soft processor. In this section, we show how the hardware monitor can seamlessly follow the execution of the TBVP applications on each core of the system. We then show how the monitor detects and defeat the attack that we introduced earlier in Section 5.3. We also show the seamless monitoring operation during

a task migration scenario in which the process starts on one core and then migrates to another and finishes its execution on the new core.

5.1 Related Work

Hardware monitoring has been proposed in the form of monitoring control-flow integrity [1]. Such an approach does not require additional hardware, but may slow down program execution and may make program execution non-deterministic, which is a problem for embedded real-time applications. Hardware monitoring at the level of basic blocs was proposed by Arora et al. [6]. This approach was improved by Mao et al. to monitor application execution at the level of individual processor instructions [41].

The need for multiple programs executing on multiple processors was first introduced by Hu et al. [32], albeit in the context of monolithic applications running on bare metal processor cores. In Chapter 2, we introduced the ability to track multiple processes managed by an operating system, but with no ability for migration [73]. We further expanded our work in Chapter 3 by introducing the ability to monitor a simplistic operating system [57]. In the same chapter, we further referenced the first hardware monitoring system that used a conventional operating system [58]. Due to the size of the operating system, this work only monitors a limited set of system calls. In this chapter, we also do not monitor the execution of the operating system, but focus on monitoring all applications. However, the previous techniques are also applicable here if necessary. The main aspect of our work that advances related work is the support for multiple processor cores and the ability to monitor under the dynamics that are introduced by an operating system. The aspects of the various prior works are summarized in Table 5.1.

Table 5.1. Related Work on Hardware Monitoring.

	Abadi et al. [1]	Arora et al. [6]	Mao et al. [41]	Hu et al. [33]	Chapter 2	Chapter 3	Provelengios et al [59]	This Chapter
verification	control flow operations	all processor instructions						
granularity	basic block		single processor instruction					
target	application / OS	monolithic application			simplistic OS		Linux OS	
coverage	application / OS	entire application			applications	entire OS	system calls	applications
overhead	software	high hardware cost					low hardware cost	
processor	single-core			multi-core		single-core		multi-core

5.2 Implementation of the Ticket-Based Verification Protocol on LEON3

The essence of TBVP is to remove the burden of communicating with a blockchain, which is the basis for economic value, from regular IoT devices (defined as *things*). In this protocol, designated *gateways* are the only devices within an administrative domain that need to be connected to the blockchain. This blockchain isolation technique enables a low-end IoT device (in terms of processing power and network bandwidth) to participate in data and economic values exchanges. Specifically, the space-time diagram of a TBVP transaction shown in Figure 5.1 shows that interactions between things do not involve direct communication with the blockchain.

We implemented the TBVP as three separate applications: *gateway*, *seller*, and *buyer*. We defined two separate administrative domains (we call them AD1 and AD2 here). In AD1, all functionalities were implemented on a desktop computer with an Intel Core i7-8700k CPU and 32GB of RAM. AD2 which implemented the system under test consisted of a desktop computer running *gateway* plus the LEON3-based embedded system running an instance of *buyer* and *seller* each in a different experimental setup. The desktop computer used in AD2 had an Intel Core i7-7700k CPU plus 16GB of RAM. We also used Ropsten (Ethereum testnet) as our blockchain platform [21].

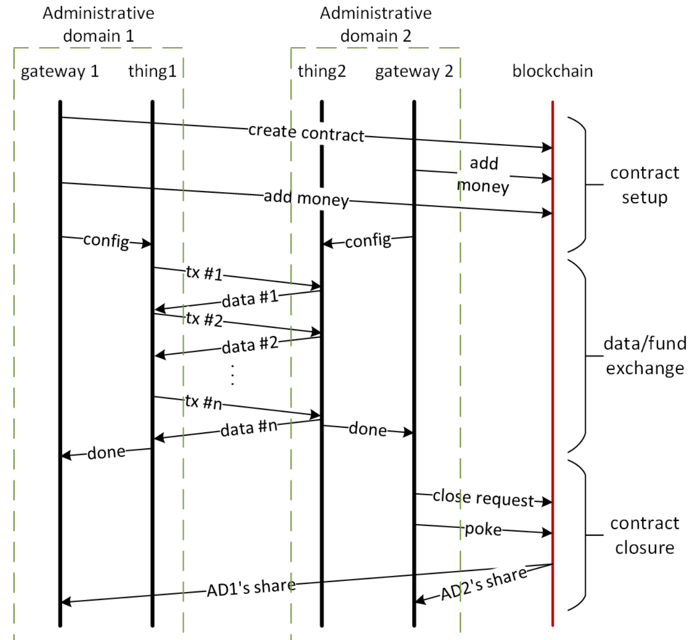


Figure 5.1. Space-Time diagram showing contract setup, data/economic-value exchange, and contract closure phases.

In our implementations, we emulated the data sources with files. We generated different files on the *seller* with random values in them. We also assumed that the Unique Resource Identifier (URI) used between the *seller* and *buyer* is a null-terminated string holding the name of the file on the *seller*.

To evaluate the correctness of our implementations, we ran two experiments. In the first experiment, we instantiated an instance of *seller* on AD1, selling 200 samples of data, each for 0.01 Ether. The initial contract set up between AD1 and AD2 had 20 Ether in it with the initial contribution of 10 Ether by each AD. The course of actions taken by the *gateways* of those two ADs, on the blockchain, is depicted in Figure 5.2.

In the first step, AD2 (with blockchain address starting with 0xc8) sets up a contract on the blockchain. The contract is funded later in steps two and three with 10 Ether from each AD. After having those two balance transfers validated by the blockchain, the systems can start sending each other signed promises spending from

	Age	From		To	Value
7-	1 min ago	0xf22ea0efdb8755a...		0xc8cccea55a9bb5...	8 Ether
	1 min ago	0xf22ea0efdb8755a...		0xf45449ca9a1b56f...	12 Ether
6-	1 min ago	0xf45449ca9a1b56f...	IN	0xf22ea0efdb8755a...	0 Ether
5-	2 mins ago	0xf45449ca9a1b56f...	IN	0xf22ea0efdb8755a...	0 Ether
4-	2 mins ago	0xf45449ca9a1b56f...	IN	0xf22ea0efdb8755a...	0 Ether
3-	5 mins ago	0xc8cccea55a9bb5...	IN	0xf22ea0efdb8755a...	10 Ether
2-	5 mins ago	0xf45449ca9a1b56f... AD1	IN	0xf22ea0efdb8755a...	10 Ether
1-	6 mins ago	0xc8cccea55a9bb5... AD2	IN	Contract Creation	0 Ether

Figure 5.2. Blockchain transactions for selling 200 data samples.

that balance locked on the contract. All those transactions are exchanged off the blockchain. Finally, after sending 200 pieces of data from AD1 to AD2, AD1 will have a signed promise from AD2 transferring the accumulated amount 2 Ether to AD1's account.

By sending this signed promise to the contract on the blockchain, AD1 updates the bookkeeping data structure of the contract on step 4. In step 5, assuming that there are no more exchanges between these two ADs, AD1 sends a *close request* to the contract. This transaction is a notice to AD2 to send a more up-to-date version of the promise between them in case the one submitted in step 4 was not the most recent. Finally, in step 6, AD1 sends a *poke* command to the contract, finalizing the closure. The closure of the contract results in a self-destruct operation and splitting the balance held by the contract according to the internal balance sheet maintained by the contract.

We can see that out of 20 Ether held by the contract, 12 Ether go to AD1 and 8 Ether to AD2 which in turn means that AD2 has paid 2 Ether to AD1. At the end of the experiment, we also checked the content of the data file retrieved by the *buyer* and made sure it was identical to the file originally stored on the *seller*. The total

amount of time taken to exchange 200 pieces of data/signed-promises was 2 minutes and 33.64 seconds.

We also ran the same experiment but with switched roles between AD1 and AD2. In this new setup, AD1 was the *seller* and the embedded system in AD2 was the *buyer*. In this experiment, the embedded system was able to retrieve the file from the *seller* successfully and in return 2 Ether were transferred from AD2 to AD1. The time it took the systems to exchange data and signed promises was 2 minutes and 25.5 seconds.

5.3 Monitoring the TBVP on a Single-core Processor Running Linux

Being a part of the data/economic-value exchange makes these IoT devices more attractive to hackers. Being connected to the Internet also makes these devices very accessible to a wide range bad actors all around the world. Hacking such a device, now, can result in leakage of private information that has economic value.

5.3.1 Attacks on Economic Transaction Protocols

When analyzing the TBVP protocol, we can see possible scenarios in which these devices can be attacked. For example, each device that acts as a buyer needs to store a copy of the private key of the blockchain account used to set up the contract between the two administrative domains (contract setup phase in Figure 5.1). By gaining access to that private key, an attacker will have full access to the Ethereum account used by that administrative domain.

On the other side of the protocol, we have the seller *things*. These entities receive promises from the buyer *things*, validate those promises, and provide the buyers with the requested data or access to IoT components. To avoid incurring fees associated with publishing a transaction on the blockchain, the seller stores the most up-to-date

promise (which has the accumulated value of all the data provided to the buyer) locally until the end of the data/economic-value exchange session. At the end of the session, the seller sends the final promise to the *gateway* of its administrative domain. Receiving the promise, the *gateway* has the option to either close the contract between the administrative domains and cash out the balance promised to it or just update its records on the balance between the two administrative domains and keep the contract open for future exchanges.

In this scenario, if the seller device crashes or loses the promise received from the buyer, there will be no way for its administrative domain to claim the money it was promised from the buyer. Thus, the buyer acquired all the data for free. One possible attack scenario is that the buyer sending promises for the data it requests until just before the end of the exchange session. Then, the buyer can exploit a potential vulnerability in the seller's system to crash the seller (e.g., with a stack smashing attack) and therefore get away with the data it has received without paying for it.

5.3.2 Attacks on Embedded Systems and Hardware Monitoring

5.3.2.1 Security Vulnerabilities in Embedded Systems

As discussed in the first chapters, there are a large number of possible attacks on embedded processors [53]. In this work, we focus on attacks that aim to change the processing behavior of the embedded system. This category of attacks is very broad and encompasses several specific attacks (e.g., attacks that use buffer overflows to smash the processing stack to redirect control-flow to malicious code). This class of attack not only applies to von-Neumann-architectures with shared instruction and data memory but also to Harvard architectures that separate these memories and are commonly used in embedded systems (e.g., "return-to-libc" attacks [12, 23]).

We assume that an attacker uses the I/O functionality of the embedded system to cause the embedded processor to misbehave. The attack could be based on an intentional modification of the processor stack (e.g., stack smashing). A controlled change in the stack makes it possible for an attacker to change the control flow such that malicious code (e.g., contained in the input if using a von-Neumann architecture) is executed (e.g., processing malicious application code). This type of attack is frequently used on the Internet to gain access to end-systems via vulnerable software. As discussed in Section 5.3.1, a cleverly timed attack of such a type can lead to insecurities when using IoT protocols that exchange economic value.

5.3.3 Hardware Monitor for IoT Protocols

In this section, we describe the design of our hardware monitor that can ensure the secure execution of IoT applications running on an embedded processor.

5.3.3.1 System Architecture

The overall architecture of the embedded system augmented with the hardware monitor is shown in Figure 5.3. In the figure, we can see the embedded processor (CPU), hardware monitor, and the secure graph loading unit. The role of the secure graph loading unit is to securely load the content of those graph memories from an outside source as the CPU starts a new application [34].

There are three main components to the hardware monitor: *instruction validation and sequencing logic (IVSL)*, *switch fabric*, and *graph memories*. The *IVSL* unit needs to have access to two pieces of information directly from the CPU: the program counter (PC) holding the address of the current instruction being executed on the CPU, and the instruction associated with that address. As shown in the figure, the datapath of the CPU is tapped to extract that information. Since those two values are tapped from the datapath, they are always available to the hardware monitor without needing the CPU to explicitly communicate them to the hardware monitor.

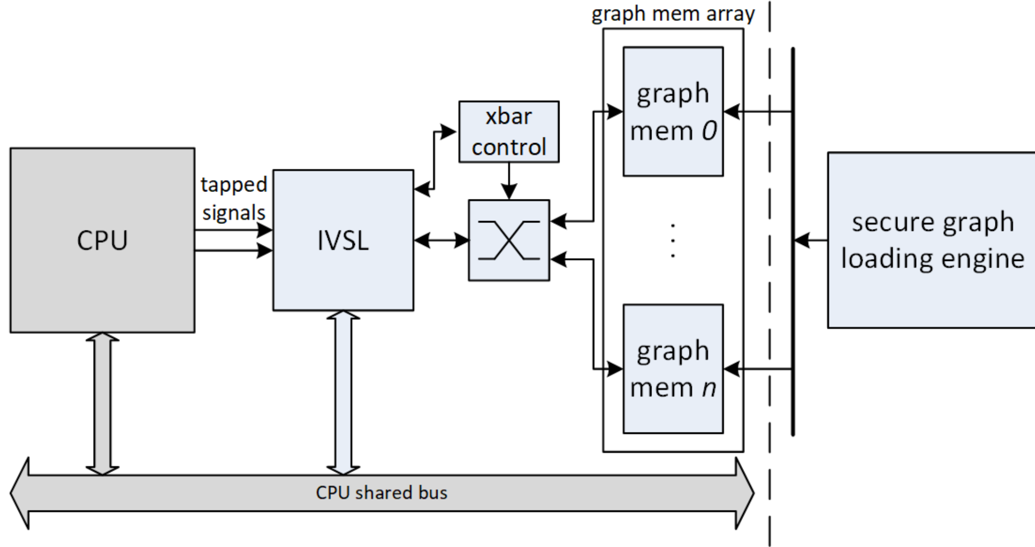


Figure 5.3. Proposed IoT system architecture with the monitoring system.

The hardware monitor is also connected to the shared bus of the CPU as a peripheral. That communication channel is used to synchronize the hardware monitor with the CPU when it comes to context-related operations (initiating a new process, switching between a process and the kernel, and terminating a process). The shared bus is also used to trigger a recovery procedure by the hardware monitor in case of detecting an attack on the CPU. The ideal behavioral model of applications is extracted in the form of monitoring graphs. The second component of the hardware monitor is an array of *graph memories*. Each of those memories holds the graph representation of an application that has to be monitored. Finally, to provide connectivity between the *IVSL* and the *graph memories*, we need a switch fabric which lies between those two entities.

5.3.3.2 Hardware Monitor Operation

During the course of its operation, the hardware monitor can be in three different states: *active*, *stopped*, and *paused*. The monitor transitions between these three states based on the execution information it receives from the CPU. It should be

noted that the actual state machine implementing the functionality has 12 states. Those 12 states, however, can be categorized under these three main categories.

Active is the state of the system in which it actively inspects every instruction executed by the CPU. In this state, the hardware monitor retrieves the graph node associated with each CPU instruction. The monitor then compares the hash value of the instruction with the permissible hash values encoded in the graph entry. In case of a match, it updates the pointer to the next graph entry based on the information available in the current graph entry and the hash value of the instruction being executed. In case of detecting a mismatch, the monitor triggers a recovery procedure on the CPU. The recovery measure can vary from one system to another based on the requirements of that particular system.

The hardware monitor is in the *stopped* state when the CPU is executing a task which does not need to be monitored. This includes the operating system itself and the tasks that are trusted or have a lower risk of being exploited. It is up to the system administrator to make that decision and mark the tasks accordingly.

Finally, the hardware monitor may need to hold off following the instructions from time to time. In such cases, the monitor enters the *paused* state. In this state, the monitor does not advance in traversing the monitoring graph as the instructions are being ignored by the CPU. One example of such cases is when the outcome of a prior control flow instruction is determined in a pipeline micro-architecture and the following instructions have been fetched from the wrong execution path. In such scenarios, all the following instructions have to be flushed out. (They pass through the pipeline without affecting the state of the system.) When in the *paused* state, the hardware monitor has to be ready to re-enter the *active* state as the CPU resumes executing the instructions from the right execution path. Different events can trigger a transition to the *paused* state. In our work, we call those events *pause triggers*.

Pause triggers vary from one CPU architecture to another and need to be chosen and treated carefully with the knowledge of the target CPU architecture.

5.3.4 Hardware Monitor Data Structure and State Transition

The monitoring lifecycle of an application starts with the operating system (OS) initiating a new process based on an executable file. If the executable is marked by the system administrator to be monitored, the OS sends the *task init* command to the hardware monitor. This command includes two essential pieces of information: process ID (PID) of the newly initiated process and the unique graph ID (GID) of the executable. Assigning a unique identifier (PID) to each process is a common practice among the modern operating systems that implement the concept of processes.

Graph ID, however, is something new which we introduce to the operating system. Each application that has been analyzed and its corresponding graph has been extracted should also be assigned a unique identifier. At runtime, this identifier is used by the hardware monitor to choose the right graph memory to retrieve the application's monitoring information. There are different ways to implement this new concept in the operating system. Possible implementations include providing a one-to-one mapping between executable file names and unique GIDs as a file to the operating system. At runtime, the OS can read the file and determine the GID associated with each executable. This mapping can also be hard-coded in the source code of the operating system when compiling it from its source code. The second approach offers more security at the expense of less flexibility.

The association between PIDs and GIDs should be recorded in the hardware monitor. The hardware monitor records those associations in a table where each entry represents an active task being monitored. For now, each entry needs to have a field for both PID and GID. Throughout this section, we incrementally add more

information to this table as we discuss the role of each piece and how it is used by the hardware monitor.

Every time the OS schedules a new task, it sends a *context switch* command along with the PID of the newly scheduled task to the hardware monitor. Using the PID to GID mapping, the hardware monitor determines which graph memory to use and sends a request for that graph memory to the *switch fabric control unit*. After acquiring access to the right graph memory, the hardware monitor has to load its internal registers. These internal registers include the 16 registers holding the starting addresses of the 16 groups of graph nodes within the graph memory. Those 16 addresses are embedded in the monitoring graph of each application.

In addition to those 16 registers, the monitoring graph also has the program counter (PC) associated with the first instruction of the application embedded in the graph. The hardware monitor uses that stored PC as a trigger to start monitoring the application. After initializing those internal registers, the hardware monitor enters the *paused* state. In this state, the monitor is waiting for the instruction address matching the triggering PC register to make the transition to the *active* state and start the monitoring operation.

Different states of the hardware monitor and the transitions between them are depicted in Figure 5.4. An important piece of information which is essential to a successful transition from the *paused* state to the *active* state is the program counter of the next instruction on the legitimate execution path. That PC can be calculated differently based on the *pause trigger* causing that state transition. For example, in case of interrupts, the PC is the current PC plus 4 (assuming that the CPU is of RISC architecture with 32-bit wide instructions). To keep track of the resuming program counter, we need to add another field to our data structure which we call it *resume PC* here.

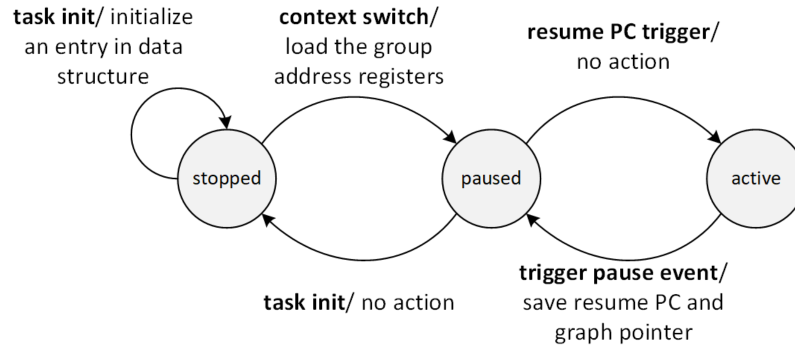


Figure 5.4. Different states of the hardware monitor and transitions between them.

The final piece of information we need to add to the data structure in the hardware monitor is a pointer to the graph memory. While in the *active* state, that pointer is calculated using the information stored in the graph memory, the hash value of the current instruction being executed, and the content of the appropriate group address register. When the CPU switches from one task to another, the state of the hardware monitor should be saved so when the CPU switches back to the current task later in the future, the hardware monitor can pick up monitoring from where it left off. That is why the data structure in the hardware monitor needs another field for the current pointer to the monitoring graph. Table 5.2 shows the data structure used by the hardware monitor to keep track of multiple applications.

5.3.4.1 Selective Monitoring

The main drawback of monitoring the applications running on an instruction-by-instruction basis is its high memory overhead. This high overhead makes this approach unsuitable for monitoring big applications. However, not all pieces of an application are equally vulnerable. That is the basis of ARM TrustZone technology [5]. In our approach, we narrow down the focus of the hardware monitoring to applications and not the operating system. Other work has addressed the challenge of monitoring the operating system [57, 58].

Table 5.2. Data structure used by hardware monitor

PID	GID	graph PTR	resume PC

Within applications, we decide on instruction-by-instruction monitoring at the granularity of functions. When extracting the monitoring graph of an application, the system administrator can mark functions as *trusted* or *untrusted*. Calling a *trusted* function from within an *untrusted* function will result in the hardware monitor making a transition from the *active* state to the *paused* state. In this case, the previous program counter plus 4 (the execution of the first instruction of the *trusted* function is a *pause trigger*) is saved in the *resume PC* field of the table entry associated with the current task. Upon returning from that *trusted* function to the *untrusted* calling function, the hardware monitor is triggered by matching the program counter on the CPU and the *resume PC* value and goes back to the *active* state and resumes monitoring the task. When marking functions, a function can be marked as *trusted* only if all the function calls inside it are to *trusted* functions. That is why the hardware monitor does not have to deal with calls from *trusted* functions to *untrusted* ones.

5.3.5 Attack Scenario: Stack Smashing Attack

In the implementation of the TBVP, when requesting a piece data, the *buyer* forms a packet including the URI of the resource it is interested in, the amount of money it is promising to the *seller*, the sequence number of the promise, and a signature. The signature is calculated based on secp256k1 [77] elliptic curve which is the standard used by both Ethereum and Bitcoin. The input to the signaure is a KECCAK-256 [8] hash of the promised amount and the sequence number packed together. Upon receiving that packet, in the *seller* application, the packet is passed to the *parse_packet* function to be parsed. In *parse_packet*, there is a local buffer for the URI string which is 64 bytes long (assuming that the maximum length of URI is

```
File  Edit  View  Search  Terminal  Help
# ./sellerLeon
valid public key!
System hijacked!
# 
```

Figure 5.5. System console after being hijacked.

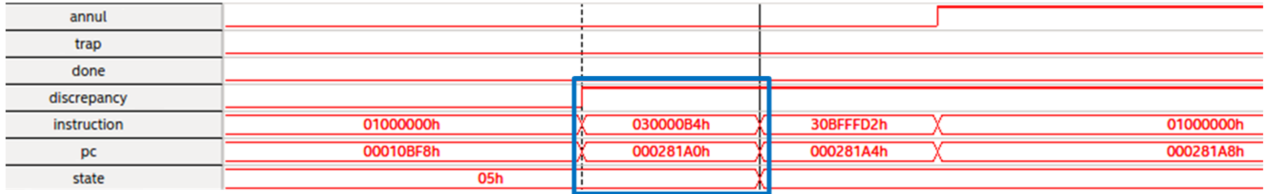
64 characters). *parse_packet* then uses the *strcpy* function to copy the URI into its local buffer. This is where the code is vulnerable. By sending a packet with a URI longer than 64 bytes, the *buyer* can overrun the stack of the *seller* and potentially override important information such as return address of the function.

In the source code of the *seller*, we have a function called *dummy_function* which is not called during the normal execution of the application. The sole function of that subroutine is to print a message: "System hijacked!" and return to the OS. By crafting a proper packet on the modified version of the *buyer* which is called *attackerBuyer* here, we managed to overwrite the return address of *parse_packet* and redirect the execution flow of the *seller* from *parse_packet* to *dummy_function* and therefore have the *seller* print the message. The system console after being hijacked is shown in Figure 5.5.

In this attack scenario, the *buyer* initiates the attack after receiving 150 samples. Normally, the *seller* should be able to send the signed promise it has received for selling 150 samples, to AD2's gateway which will then be used to earn AD2, 1.5 Ether. However, since the execution of the *seller* was derailed by the attack, that promise never made it to the *gateway*. As a result, when AD1 closes the contract, we observe that both ADs receive an equal amount of 10 Ether each.

5.3.6 Attack Detection and Recovery

In the next step, we added the hardware monitor to the embedded system. We used Quartus SignalTap to extract the internal signals of the FPGA. We extracted



(a) Signals extracted from the FPGA

	Age	From		To	Value
7-	47 secs ago	0xe89057ea3831ef5...		0xc8cccea55a9bb5...	8.5 Ether
	47 secs ago	0xe89057ea3831ef5...	→	0xf45449ca9a1b56f...	11.5 Ether
6-	47 secs ago	0xf45449ca9a1b56f...	IN	0xe89057ea3831ef5...	0 Ether
5-	4 mins ago	0xf45449ca9a1b56f...	IN	0xe89057ea3831ef5...	0 Ether
4-	4 mins ago	0xf45449ca9a1b56f...	IN	0xe89057ea3831ef5...	0 Ether
3-	6 mins ago	0xf45449ca9a1b56f...	IN	0xe89057ea3831ef5...	10 Ether
2-	6 mins ago	0xc8cccea55a9bb5...	IN	0xe89057ea3831ef5...	10 Ether
1-	9 mins ago	0xc8cccea55a9bb5...	IN	Contract Creation	0 Ether

(b) Blockchain transactions after defeating an attack

Figure 5.6. (a) Signals extracted from the FPGA board showing the detection of the attack. (b) Resulting blockchain transactions after detecting the attack.

the monitoring graphs of both *seller* and *buyer* applications and loaded them on the hardware monitor. We ran both scenarios described in Section 5.2 and observed that the hardware monitor does not raise any false positive flags.

In Section 5.3.1, we discussed how in the case of the *seller* application, closing the application under attack is also beneficial to the attacker. To address that, we came up with a novel approach to dealing with attacks. We added a *recovery* function to the *seller* application. This recovery application is a signal handler registered with the Linux kernel. Upon receiving an attack signal from the hardware monitor, the kernel is interrupted. We developed an Interrupt Service Routine (ISR) to handle the attack cases. This ISR receives the PID of the application under attack from the hardware monitor and sends a recovery signal to that application invoking the *recovery* function.

Upon receiving the signal, the *recovery* function retrieves the last validated promise, sends that validated promise to the *gateway*, and exits. This way, the *gateway* has the chance to claim the amount of money the *seller* was promised before being attacked. The *gateway* then submits the promise to the blockchain and follows the appropriate steps to close the account and receive the funds. Figure 5.6(a) shows the signals extracted from the FPGA. It shows how the hardware monitor detects the attack once the CPU starts to execute instructions from the *dummy_function* that are not a legitimate sequence of instructions that can follow the last instruction of *parse_packet*. The blue box in the figure shows how the moment the first instruction in the *dummy_function* (instruction binary value of 0x030000B4h) at the address 0x000281A0h is executed on the CPU, the discrepancy flag is raised in the hardware monitor. Figure 5.6(b), subsequently, shows how the promise received by the gateway of AD2 is used to claim the funds on the contract (step 4). It also shows how the 20 Ether on the contract is split between the two parties accordingly (step 7).

5.4 Monitoring the TBVP on a Multi-core Processor

In the previous section, we demonstrated how a vulnerable implementation of the TBVP can be exploited in the absence of any protection mechanism. We showed how such runtime attacks can be detected and defeated using our proposed hardware monitoring solution. In this section, we expand that work to multi-core systems. We go over the challenges we face when expanding the hardware monitoring approach to a multi-core system and how we address those challenges. We describe the design details of our hardware monitor and show how it can seamlessly integrate with a multi-core LEON3 processor. Evaluating our prototype system, at the end of this chapter, we show that the hardware monitor can follow the execution of an application on different cores and follow them when migrating from one core to another with minimum performance slowdown.

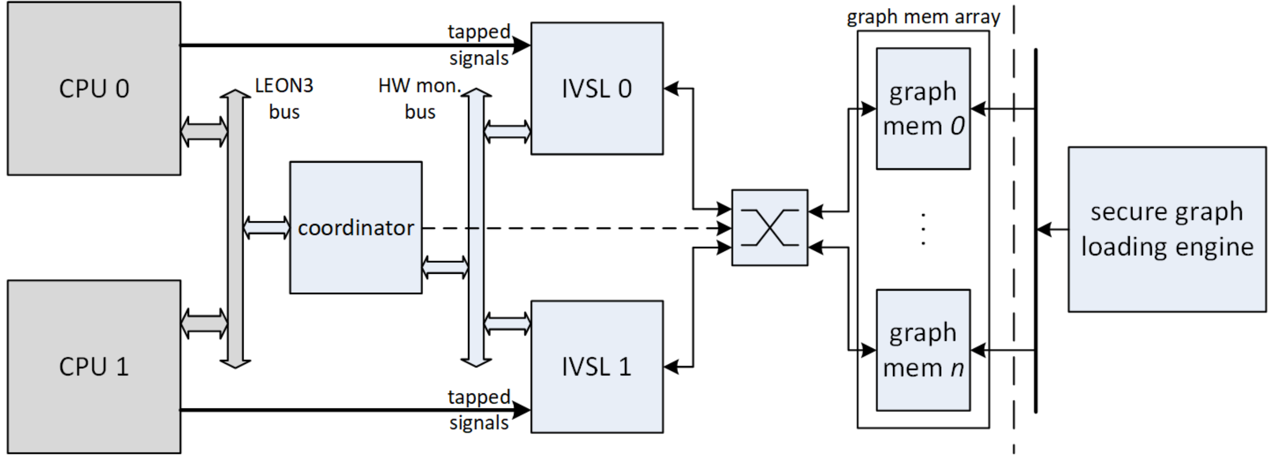


Figure 5.7. Proposed system architecture for a multi-core LEON3 processor augmented with a hardware monitor.

5.4.1 System Architecture

The proposed system architecture for a multi-core LEON3 processor, augmented with a hardware monitor, is shown in Figure 5.7. This new architecture can be seen as an extension to the architecture for a single-core system that we proposed earlier in this chapter (shown in Figure 5.3). In the multi-core architecture, the *IVSL* unit has been replicated to match the number of cores. These *IVSL* units are no longer connected to the shared bus of the cores. There is a new intermediary node, called *coordinator*, added between the CPU cores and the *IVSLs*. The role of the *coordinator* is to coordinate between the OS running on the CPU cores and the *IVSLs*. In Section 5.4.2 we describe the functionality of this module in more detail.

The *coordinator* module is connected to *IVSLs* via an internal bus (it is not accessible to the CPU cores, both for security and performance considerations). The bus is used for three different purposes: 1- to communicate both commands and context-related information between the *coordinator* and the *IVSLs* when a new context related event takes place on the CPU; 2- to notify the *coordinator* about an attack being detected by the *IVSLs*; and 3- to transfer monitoring state between *IVSLs* in case of task migration. The last modification to this model compared to the

Table 5.3. Data structure used by the coordinator

PID	GID	CPU ID

singlecore system is the removal of the *switch fabric control* module. The scheduling of the switch fabric is now done by the *coordinator*.

5.4.2 Hardware Monitor Operation and Data Structure

The hardware monitoring operation starts with a new task being executed by the operating system. In the *execve()* system call, inside the kernel, the application name is compared against a list of applications marked by the system administrator for monitoring. In case the application being executed is in the list, the unique graph identifier associated with that application (in that list) along with the PID of this new process and the core ID of the core running it, is communicated to the *coordinator* in the hardware monitor. As we have multiple cores, each one of the cores independently can run *execve()* and therefore they will end up in a race condition.

To enforce mutual exclusion when communicating with the hardware monitor, we restricted the communication with the hardware monitor to only take place through the hardware monitor driver. In the driver, the mutual exclusion is enforced using spin_locks which implement busy waiting locking mechanism only appropriate for multi-core systems [39].

5.4.2.1 Task Initiation

Receiving the *task_init* command from the CPU, the *coordinator* makes a new record in its bookkeeping table, shown in Table 5.3, associating the reported PID, GID, and core ID. It then forwards that command to the appropriate *IVSL* via the internal bus. From this point on, the monitoring takes place on that particular *IVSL* without any interventions by the *coordinator* until the next-context related operation takes place on the CPU or one of the *IVSLs* detects a deviation.

5.4.2.2 Context Switch

Context switch is the second context-related operation on the OS that triggers communication between the CPU and the monitor. Context switch happens in the *__schedule()* function inside the kernel. We modified this function to report the PID of the process it is switching to along with the core ID on which it is scheduling the process to the hardware monitor. Receiving the *context_switch* command, the *coordinator* first consults its table to see if the reported PID is a match for any of the PIDs currently being monitored. If it is not a match, it means that the process does not need to be monitored. However, if the reported PID matches one of the PIDs in the table, the *coordinator* has to trigger the context switch procedure on the appropriate *IVSL*.

In case of a match, the next step would be checking the core ID of this newly scheduled process against the current *IVSL* responsible for monitoring that process. If the process is resuming on the same core, then the associated *IVSL* is triggered to perform the context switch and prepare for resuming the monitoring. Alternatively, if the reported core ID does not match the recorded *IVSL*, it means that the process has migrated to another core and a process migration procedure should be performed.

5.4.2.3 Process Migration

Process migration is triggered when the CPU schedules a process on a core different from the one it was running on before. In this case, the *coordinator* sends a *record_retrieve* command to the *IVSL* which was previously monitoring that process to share the records of monitoring that process on the internal bus. This command has 3 fields: *source_IVSL ID*, *destination_IVSL ID*, and *PID* of the migrating process.

After issuing the command, being the bus master, the *coordinator* grants bus access to the source *IVSL*. Upon finding the communicated PID in its data structure, the source *IVSL* puts the GID, graph PTR, and resume PC fields of the entry associ-

ated with that PID on the internal bus and asserts its *data_ready* signal. Seeing that *data_ready* signal, the destination *IVSL* picks up the data from the bus and updates the entry it has already created (upon seeing its ID on the original command from the *coordinator*) in its data structure for that PID. It then automatically triggers a *context_switch* operation locally. The source *IVSL* putting the record on the internal bus also lets the *coordinator* know that the transfer was successful and it can update its data structure as well and eventually signal the OS the end of its *context_switch* procedure.

5.4.3 Attack Detection and Recovery

An ongoing attack is detected when one of the *IVSLs* detects a deviation from the ideal model in one of the processes on the system. The same mechanism proposed in Section 5.3.6 can still be used here. However, since here we have multiple cores we have to decide which core has to serve the interrupt. On the other had, one of the promises of monitoring the applications on an instruction-by-instruction basis is to detect and stop the attack, ideally, once the first attack instruction runs on the CPU.

To speed up the attack recovery in the kernel, instead of having one interrupt signal going from the hardware monitor to the interrupt controller, we reserve multiple interrupt lines for the hardware monitor. The coordinator uses each of those lines for one of the *IVSLs*. On the other side, in the driver, each interrupt is masked by all the cores but the one associated with that particular *IVSL*. By doing this only one core will serve an interrupt and that core will be the one under attack. This approach makes sure the core under attack is interrupted as soon as the deviation is detected by the *IVSL* and it also does not interrupt the execution of other processes on the other cores.

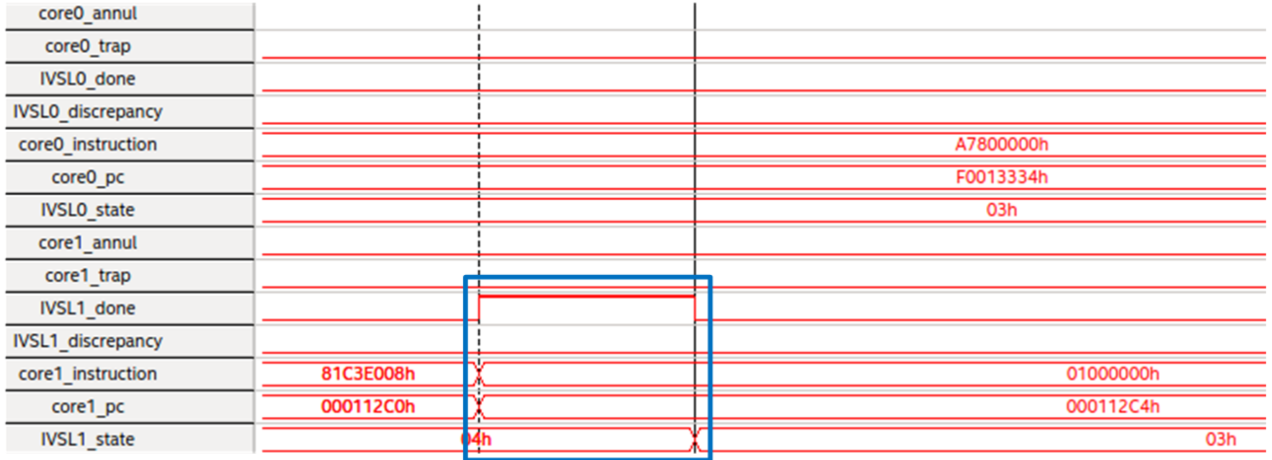


Figure 5.8. Waveform extracted from the FPGA showing the Completion of a task on core 1.

5.4.4 Implementation and Evaluation

To evaluate this new system, we again defined two separate administrative domains, AD1 and AD2. In AD1, all functionalities were implemented on a desktop computer with an Intel Core i7-8700k CPU and 32GB of RAM. AD2 which implemented the system under test consisted of a desktop computer running *gateway* plus the LEON3-based embedded system running our embedded applications. The desktop computer used in AD2 had an Intel Core i7-7700k CPU plus 16GB of RAM. For the exchanged data, we similarly emulated the data sources with files.

5.4.4.1 Normal Execution on One Core

In the first step, we have to make sure that this new hardware monitor can be properly integrated with the multi-core CPU. We ran the *seller* and *buyer* applications multiple times to make sure the can run smoothly on the CPU without any false alarms from the hardware monitor. Checking the behavior of the applications, we observed that they run with no problem and perform their tasks resulting in the transfer of funds on the blockchain.

We also extracted the signals from the hardware monitor to make sure it is following the execution of the tasks throughout their entire execution life-cycle. Figure 5.8 shows the completion of the monitoring task for the *seller* application. In this run, the kernel scheduled the *seller* application on core1. As shown in the highlighted area of the figure , once the CPU executes the instruction at address 0x000112c4, which is the last instruction in that application, the monitor matches that instruction with the graph of that application and raises the *job_done* flag. This flag does not serve any purpose in the internal structure of the *IVSL* and is only used for debugging purposes.

5.4.4.2 Continuous Monitoring In Presence of Task Migration

In Section 5.4.2.3, we described how the hardware monitor can follow the task migration procedure in the kernel without losing track of the application execution on the CPU. To verify the correct behavior of the hardware monitor, we ran two instances of the *seller* application on the FPGA board in AD2. As the application, initially has to wait for an incoming connection, the application status changes from ready to waiting right after it start. As a result of that status change, the kernel removes the associated process from the kernel ready queue as the process invokes a system call to listen for an incoming connection. We believe that to be the reason why both instances of the application were originally scheduled on one core (in this example, core 1) instead of being distributed over both of the cores.

Next, we ran two instances of the *buyer* application in AD1 and had them connect to the waiting instances of the *seller* application in AD2. We noticed that once the connections are established, the kernel re-balances the load on the cores by moving one *seller* process to another core. We captured the operation of the hardware monitor following a task migration in the kernel using SignalTap. The extracted waveform is shown in Figure 5.9.

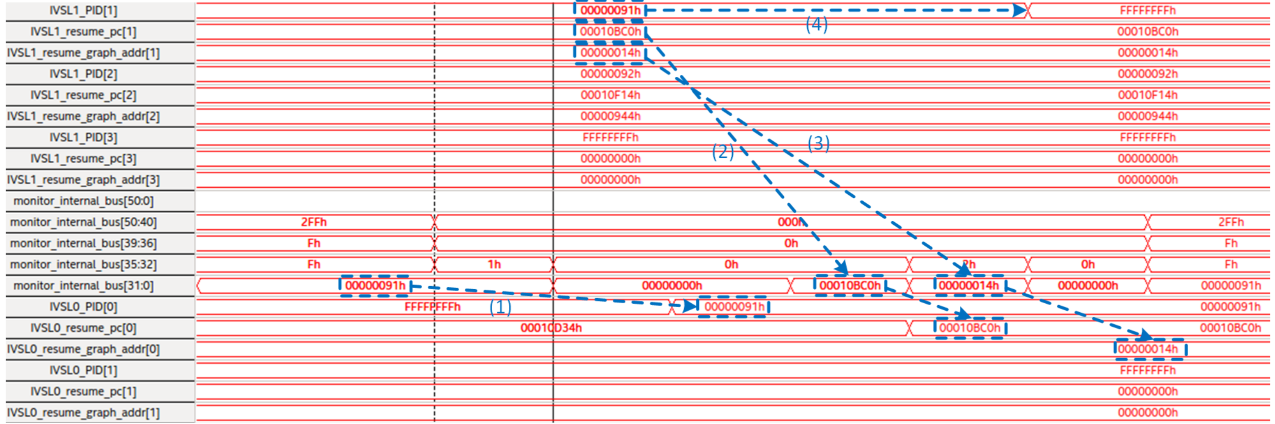


Figure 5.9. Waveform extracted from the FPGA showing the successful transfer of monitoring state across two IVSLs.

In the figure, we have highlighted four significant steps it takes for the monitor to follow the task migration. In the first step, the coordinator puts the PID of the process migrating along with the source core and the destination core on the *monitor_internal_bus*; Bits 0 through 31 represent the PID; Bits 32 through 35 represent the source core (and therefore the source IVSL) while bits 36 through 39 determine the destination core of the migrating process. Upon seeing its ID on the *monitor_internal_bus*, IVSL0 initializes an entry in its bookkeeping table with the PID of the migrating task and waits for the source IVSL (in this case IVSL1) to provide the rest of the monitoring state. In the figure, we can see that the PID of the migrating task, 0x00000091h is being loaded into the first entry of the bookkeeping table on IVSL0. *IVSL0_PID[0]* is the PID field of the first entry in the bookkeeping table and it is loading the PID of the migrating process.

In step two, the source IVSL, in this case IVSL1, takes over the internal bus. After acquiring the control of the bus it puts the *resume_pc* of the migrating process on the bus which is subsequently loaded in the *resume_field* of the newly created entry in the IVSL0's bookkeeping table. We can see the value of 0x00010BC0h being transferred on the bus and subsequently loaded into the *IVSL0_resume_pc[0]*.

Step three shows the transfer of the last piece of information from IVSL1 to IVSL0, *resume_graph_addr*. In the figure, we can see the transfer of the value 0x00000014h from the IVSL1 to the internal bus and from the internal bus to IVSL0.

The process migration procedure is done in the hardware monitor as IVSL1 removes the entry associated with the migrating task from its bookkeeping table. It is done by changing the PID field of that entry from 0x00000091h to 0xffffffff which is used to mark vacant entries.

5.4.4.3 Attack Detection and Recovery

To check the capability of the system to detect and defeat runtime attacks, we implemented the attack scenario described in Section 5.3.5. We had the *buyer* application send the signed promises to the *seller* for the first 150 samples. For the 151st sample, the *buyer* sends a message that causes a buffer overflow in the *seller* application and subsequently redirects the exection flow of that application to the *dummy_function*.

Figure 5.10 shows how *IVSL0* detects the attack once the CPU starts to execute instructions inside the *dummy_function*. In the highlighted part of the figure, we can see that the *IVSL0* raises its discrepancy flag once the CPU start the execute the first instruction of the *dummy_function* at address 0x000281A0h. After detecting the attack by the hardware monitor, the recovery procedure is triggered in the kernel. As a result of the recovery procedure, AD2 closes the blockchain contract and gets compensated for the 150 samples of data it has sold. The blockchain transactions are similar to the ones shown in Figure 5.6(b).

5.4.4.4 Resource Utilization and Hardware Overhead of the Proposed Hardware Monitor

We implemented a prototype of our system on an Altera DE4 FPGA board. The synthesis and fitting report of the system is summarized in Table 5.4. The amount of

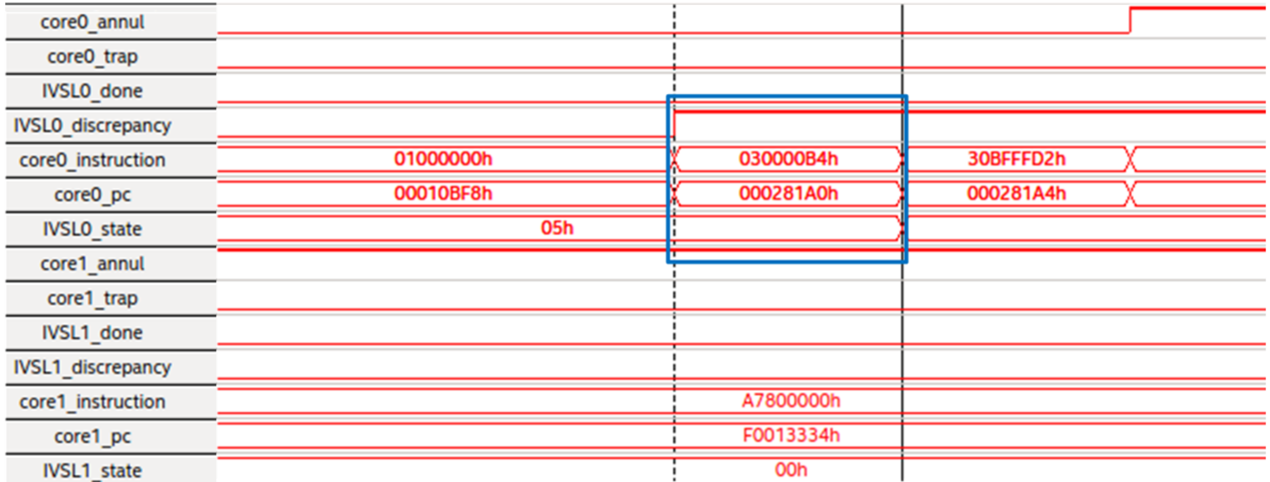


Figure 5.10. Waveform extracted from the FPGA showing the detection of an attack on core 0.

combinational resources needed by the hardware monitor is about 7% of the resources required to implement the multi-core LEON3-based system. The percentage is slightly less than 12% when we look at the registers used to realize the sequential circuits of the design. Both 7% and 12% hardware overhead are reasonable given the degree of security this approach offers. When looking at the block memory bits (onchip memory resources) used by the hardware monitor, we can see an overhead of 68%.

That is because in order to be able to keep up with the CPU, the hardware monitor needs to have the graphs stored in very fast memories. In this implementation, the same type of memory that is used for the cores' L1 cache was used for graph memories as well. The assumption here is that the graphs are located on the secondary storage and can be loaded into those memories on demand. In this setup since we emulated the secondary storage in the RAM, we could not extract any performance numbers that can be close to what we should expect in a real-world setup with a flash-based disk or a magnetic hard drive.

Table 5.4. FPGA resources consumed by different entities.

entity	submodule	combinational ALUTs	dedicated logic registers	block memory bits
LEON3	core 0	7754	5585	364000
	core 1	7802	5586	364000
	IRQ controller	353	153	0
	memory controller	161	162	0
	ethernet mac 0	2841	1824	22055
	ethernet mac 1	2762	1816	22055
	DDR controller	2862	3795	110119
	UART module	202	181	0
	other	2160	1170	256
	total	26897	20272	882485
hardware monitor	coordinator	173	272	0
	IVSL 0	725	1126	320
	IVSL 1	692	1034	264
	crossbar	274	0	0
	graph memory 0	0	0	303104
	graph memory 1	0	0	303104
	total	1864	2432	606792
debugging system	signal tap	1593	13934	145664
	debug support unit	824	341	16384
	total	2417	14275	162048

5.4.4.5 Performance Overhead

The key to successful integration of the hardware monitor and the embedded system is the communication between the kernel and the coordinator module. That communication requires the kernel to run extra lines of code. The kernel also has to wait for the hardware monitor to catch up with the context-related operations from time to time. To measure the overhead of the added communication, we ran our IoT applications in different sets of experiments first with no hardware monitor added to the system and another time with the hardware monitor present.

In one set of experiments, we ran *seller* to sell 10, 50, 100, and 500 samples. For each number of samples, we ran the experiment 10 times. The average time it took for the application to finish selling those number of samples is summarized in Table 5.5.

In another set of experiments, we ran *buyer* to buy 10, 50, 100, and 500 samples. The average time it took the *buyer* application to buy those number of samples is also reflected in Table 5.5. The percentage shown in parenthesis under the system column reflects the time overhead added by the kernel compared to the application

Table 5.5. Time spent in the user application versus the kernel when running different workloads for different system configurations. (values are in seconds)

		buyer		seller	
samples	config	user	system	user	system
10	w/ monitor	7.29	0.04 (0.54%)	7.67	0.04 (0.52%)
	w/o monitor	7.26	0.03 (0.37%)	7.49	0.03 (0.36%)
50	w/ monitor	35.86	0.12 (0.33%)	38.21	0.07 (0.18%)
	w/o monitor	36.18	0.11 (0.30%)	37.56	0.06 (0.16%)
100	w/ monitor	72.88	0.25 (0.34%)	76.37	0.14 (0.18%)
	w/o monitor	72.35	0.19 (0.26%)	75.22	0.10 (0.13%)
500	w/ monitor	362.77	1.23 (0.33%)	375.43	0.50 (0.13%)
	w/o monitor	362.02	1.07 (0.29%)	375.78	0.41 (0.11%)

itself. By adding the hardware monitor to the system, the time spent in the kernel almost doubles for short runs of the applications (10 samples). As the number of samples increases the effect of adding the hardware monitor diminishes. It can be due to the high overhead of task initiation procedure in the hardware monitor. Since task initiation happens only once and after that, the only context-related operation in the OS is context switch, when running the application for long periods, that high overhead is amortized over a longer time and therefore its effect diminishes.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

The main focus of this work was on trustworthy IoT systems. We started this work by focusing on the hardware platform of IoT devices. We believed that the fast-paced nature of IoT market prevents application developers from spending enough time on different aspects of their products. One important aspect of an IoT application which is often overlooked is the security of that application. The numerous attacks on IoT systems that we see regularly are strong evidence of that claim. At the time, the idea of hardware monitoring of embedded systems was introduced in the literature but the target devices were very limited in terms of resources and capabilities and by no means were good representatives of the embedded devices used in commercial IoT applications.

One aspect of our work was to bridge the gap between the commercial IoT devices and the hardware monitoring solutions for embedded systems. In the work presented in Chapter 2, we took the first step by introducing our novel hardware monitor that was able to follow the execution of multiple applications running on a μC/OS-II operating system. This work was the first work to integrate a hardware monitor with a real-time operating system. We showed the effectiveness of our approach to detect and defeat a runtime attack by prototyping the system on an FPGA board. We used an Altera DE4 FPGA board implementing a NIOS II processor. The overhead of our system was about 30% in terms of hardware resources and the performance slow down was negligible.

After successfully monitoring multiple application in an OS-based environment, we expanded our work to address the potential vulnerabilities in the OS as well as in the applications. The work presented in Chapter 3, expanded our previously proposed hardware monitoring system to include monitoring the OS as well. In this work, we addressed the challenges of seamless integration of the hardware monitor with the embedded processor in the absence of explicit directions from the OS.

We showed the effectiveness of our system in detecting and defeating a format string attack by prototyping it on an Altera DE4 FPGA board. We used a NIOS II processor running a μC/OS-II operating system. The evaluation of the prototype showed that this new hardware monitor adds 25% overhead to the NIOS II system in terms of hardware resources and the performance slow down is still negligible. On the other hand, as the code base of the OS is much bigger than a regular embedded application, the memory overhead of the system starts to become an issue. That is why we believe that monitoring the entire OS for more general OSes is not practical and the monitoring should be done more selectively on parts of the OS or the applications that are deemed more vulnerable.

In Chapter 4, we shift our focus from the underlying hardware platform the protocols used in IoT applications. The use of blockchain had enabled IoT devices to monetize their exchanges utilizing this technology. However, the processing and network bandwidth overhead of running a blockchain client along with its heavy reliance on the operating system and library support were the main barriers in expanding the use of blockchain technology to low-end and mid-range IoT devices.

In that chapter, we proposed our Ticket-Based Verification Protocol which separates the functionality of an IoT device into *blockchain-based* and *blockchain-agnostic* operations. By introducing this separation, our TBVP enables the low-level and mid-range IoT devices to also participate in the financial exchanges based on the blockchain without the need to directly communicate with it and therefore avoid

relying on a sophisticated OS or requiring a high network bandwidth. We then evaluated our proposed protocol by prototyping that using commercial IoT devices such as Raspberry Pi and TI CC3200 Launchpad boards. Our evaluations showed that the devices can successfully participate in financial transactions on Ethereum blockchain and the performance shown by them was reasonable given the processing power of them.

The last piece of work presented in this dissertation was on achieving a trustworthy IoT system both on protocol and hardware levels. This piece which was presented in Chapter 5 brought together the ideas of hardware monitoring and secure IoT protocols. In this chapter, we implemented the IoT protocol that we had proposed earlier on an embedded system that was representative of modern IoT devices. Initially, we explored the challenges of monitoring IoT applications running on a full-scale Linux kernel managing a 7-stage pipeline LEON3 CPU. We further expanded that work to monitor the applications running on a multi-core processor. We evaluated both the effectiveness and performance of our proposed solution by prototyping it on an Altera DE4 board.

Our evaluations showed that the system can successfully detect and defeat a real-world stack smashing attack. The overhead imposed by this hardware monitoring approach was about 7% for combinational resources it consumes and around 12% for the sequential resources.

The main drawback of our proposed hardware monitoring approach is the big memory footprint of the monitoring graphs. The number of graph entries in a monitoring graph is proportional to the number of instruction in the application binary. Storing the graph on on-chip memories is necessary if the hardware monitor is to keep up with the execution of the application on the CPU. Having big on-chip memories increases the cost of the embedded system which is highly undesirable.

We believe the idea of the memory hierarchy is applicable in this context as well. We can have secondary storage units holding the entire graph which a fast on-chip memory caches a subset of graph entries that are going to be accessed in near future. However, we cannot see the spatial locality and temporal locality in the pattern of accesses to the graph memory. The lack of locality in the access pattern of graph memories makes old caching approaches not helpful in speeding up these accesses. We believe a deeper analysis of the access patterns can be a good research topic for future projects.

In this work, we used selective monitoring approach which enabled us to only monitor parts of the code that are more vulnerable and skip monitoring the less vulnerable parts. By employing this technique, we were able to monitor IoT applications despite their big memory footprint and therefore big corresponding graphs. In this work, we assumed that the system administrator can decide which functions to trust and which ones to monitor. However, we did not talk about how these decisions should be made. We believe a more methodical way to analyze application binaries and mark the functions as *trusted* and *not trusted* can be another future direction to pursue. The outcome of that project can be an automated or assisted analysis of the application binary and generating the monitoring graph of that code.

BIBLIOGRAPHY

[1] Abadi, Martín, Budiu, Mihai, Erlingsson, Úlfar, and Ligatti, Jay. Control-flow integrity principles, implementations, and applications. In *ACM Conference on Computer and Communications Security* (Alexandria, VA, Nov. 2005), pp. 340–353.

[2] Amazon. Amazon web services. `https://aws.amazon.com/`.

[3] Antonakakis, Manos, April, Tim, Bailey, Michael, Bernhard, Matt, Bursztein, Elie, Cochran, Jaime, Durumeric, Zakir, Halderman, J Alex, Invernizzi, Luca, Kallitsis, Michalis, et al. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 1093–1110.

[4] ARM. Arm classic processors. https://developer.arm.com/products/processors/classic-processors.

[5] ARM Ltd. ARM TrustZone.

[6] Arora, Divya, Ravi, Srivaths, Raghunathan, Anand, and Jha, Niraj K. Secure embedded processing through hardware-assisted run-time monitoring. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)* (Munich, Germany, Mar. 2005), pp. 178–183.

[7] Bahga, Arshdeep, and Madisetti, Vijay K. Blockchain platform for industrial internet of things. *Journal of Software Engineering and Applications 9*, 10 (Oct. 2016), 533–546.

[8] Bertoni, Guido, Daemen, Joan, Peeters, Michaël, and Van Assche, Gilles. Keccak specifications. *Submission to nist (round 2)* (2009), 320–337.

[9] Bitcoin. https://bitcoin.org/en/.

[10] bitcoinj. Working with micropayments. https://bitcoinj.github.io/working-with-micropayments, 2017.

[11] Buterin, Vitalik. A next-generation smart contract and decentralized application platform. White paper, Ethereum Wiki, Dec. 2014.

[12] Chandrikakutty, Harikrishnan, Unnikrishnan, Deepak, Tessier, Russell, and Wolf, Tilman. High-performance hardware monitors to protect network processors from data plane attacks. In *Proc. of 50th Design Automation Conference (DAC)* (Austin, TX, June 2013), pp. 80:1–80:6.

[13] Chandrikakutty, Harikrishnan, Unnikrishnan, Deepak, Tessier, Russell, and Wolf, Tilman. High-performance hardware monitors to protect network processors from data plane attacks. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE* (2013), IEEE, pp. 1–6.

[14] Chasaki, Danai, and Wolf, Tilman. Attacks and defenses in the data plane of networks. *IEEE Transactions on Dependable and Secure Computing 9*, 6 (Nov. 2012), 798–810.

[15] Chiueh, Tzi-Cker, and Hsu, Fu-Hau. Rad: a compile-time solution to buffer overflow attacks. In *Proc. of 21st International Conference on Distributed Computing Systems (ICDSC)* (Apr. 2001), pp. 409–417.

[16] Christidis, Konstantinos, and Devetsikiotis, Michael. Blockchains and smart contracts for the internet of things. *IEEE Access 4* (May 2016), 2292–2303.

[17] Conoscenti, Marco, Vetro, Antonio, and De Martin, Juan Carlos. Blockchain for the internet of things: A systematic literature review. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)* (2016), IEEE, pp. 1–6.

[18] Cowan, Crispin, Barringer, Matt, Beattie, Steve, Kroah-Hartman, Greg, Frantzen, Michael, and Lokier, Jamie. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (Washington, DC, Aug. 2001).

[19] Devietti, J., Blundell, C., Martin, M., and Zdancewic, S. Hardbound: Architectural support for spatial safety of the c programming language. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2008).

[20] Ethereum. https://www.ethereum.org/.

[21] etherscan. TESTNET Ropsten.

[22] Federal Trade Commission. *Internet of Things: Privacy & Security in a Connected World*, Jan. 2015.

[23] Francillon, Aurélien, and Castelluccia, Claude. Code injection attacks on Harvard-architecture devices. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CSS)* (Alexandria, VA, Oct. 2008), pp. 15–26.

[24] Fruhlinger, Josh. The mirai botnet explained: How teen scammers and cctv cameras almost brought down the internet. `https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-` `html`, March 2018. Accessed: 2018-11-30.

[25] Gartner Inc. https://www.gartner.com/en/newsroom.

[26] Google. Google cloud iot. https://cloud.google.com/solutions/iot/.

[27] Gubbi, Jayavardhana, Buyya, Rajkumar, Marusic, Slaven, and Palaniswami, Marimuthu. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems 29*, 7 (Sept. 2013), 1645–1660.

[28] Guthaus, Matthew R., Ringenberg, Jeffrey S., Ernst, Dan, Austin, Todd M., Mudge, Trevor, and Brown, Richard B. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Annual Workshop on Workload Characterization* (Austin, TX, Dec. 2001).

[29] Hees, Heiko. Raiden network: Off-chain state network for fast DApps. In *Devcon Two* (Shanghai, China, Sept. 2016), Ethereum Foundation.

[30] Heilman, Ethan, Baldimtsi, Foteini, and Goldberg, Sharon. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In *International Conference on Financial Cryptography and Data Security* (2016), Springer, pp. 43–60.

[31] Hu, Kekai, Chandrikakutty, Harikrishnan, Goodman, Zachary, Tessier, Russell, and Wolf, Tilman. Dynamic hardware monitors for network processor protection. *IEEE Transactions on Computers 65*, 3 (Mar. 2016), 860–872.

[32] Hu, Kekai, Chandrikakutty, Harikrishnan, Tessier, Russell, and Wolf, Tilman. Scalable hardware monitors to protect network processors from data plane attacks. In *Proc. of First IEEE Conference on Communications and Network Security (CNS)* (Washington, DC, Oct. 2013), pp. 314–322.

[33] Hu, Kekai, Chandrikakutty, Harikrishnan Kumarapillai, Goodman, Zachary, Tessier, Russell, and Wolf, Tilman. Dynamic hardware monitors for network processor protection. *IEEE Transactions on Computers 65*, 3 (2015), 860–872.

[34] Hu, Kekai, Wolf, Tilman, Teixeira, Thiago, and Tessier, Russell. System-level security for network processors with hardware monitors. In *Proc. of 51st Design Automation Conference (DAC)* (San Francisco, CA, June 2014), pp. 211:1–211:6.

[35] Huckle, Steve, Bhattacharya, Rituparna, White, Martin, and Beloff, Natalia. Internet of things, blockchain and shared economy applications. *Procedia Computer Science 98*, Supplement C (2016), 461–466. The 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)/The 6th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2016)/Affiliated Workshops.

[36] Jaeger, Trent, Ed. *Operating System Security.* Morgan and Claypool, 2008.

[37] Jim, Trevor, Morrisett, J. Greg, Grossman, Dan, Hicks, Michael W., Cheney, James, and Wang, Yanling. Cyclone: A safe dialect of C. In *Proc. of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)* (Monterey, CA, June 2002), pp. 275–288.

[38] Jonker, Mattijs, King, Alistair, Krupp, Johannes, Rossow, Christian, Sperotto, Anna, and Dainotti, Alberto. Millions of targets under attack: A macroscopic characterization of the DoS ecosystem. In *Proceedings of the Internet Measurement Conference (IMC)* (London, United Kingdom, Nov. 2017), pp. 100–113.

[39] Kernel.org. Spin locks. `https://www.kernel.org/doc/Documentation/locking/spinlocks.txt`.

[40] Litecoin. https://litecoin.org/.

[41] Mao, Shufu, and Wolf, Tilman. Hardware support for secure processing in embedded systems. In *Proc. of 44th Design Automation Conference (DAC)* (San Diego, CA, June 2007), pp. 483–488.

[42] Mao, Shufu, and Wolf, Tilman. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers 59*, 6 (June 2010), 847–854.

[43] Medaglia, Carlo Maria, and Serbanati, Alexandru. An overview of privacy and security issues in the internet of things. In *The Internet of Things*. Springer, 2010, pp. 389–395.

[44] Microsoft. Microsoft azure. https://azure.microsoft.com/en-us/.

[45] Mogul, Jeffrey C. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Conference Proceedings* (Baltimore, MD, June 1989), pp. 203–221.

[46] Moscola, James, Lockwood, John, Loui, Ronald P., and Pachos, Michael. Implementation of a content-scanning module for an Internet firewall. In *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)* (Napa, CA, Apr. 2003), p. 31.

[47] Motorola Inc. https://www.motorola.com/us/home.

[48] Nagarakatte, Santosh, Zhao, Jianzhou, Martin, Milo, and Zdancewic, Steve. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. of the International Conference on Programming Language Design and Implementation* (June 2009).

[49] Nakamoto, Satoshi. *Bitcoin: A peer-to-peer electronic cash system*, 2008.

[50] National Institute of Standards and Technology. *National Vulnerability Database*. `http://nvd.nist.gov`.

[51] Neuman, B Clifford, and Ts'o, Theodore. Kerberos: An authentication service for computer networks. *IEEE Communications magazine 32*, 9 (1994), 33–38.

[52] Omohundro, Steve. Cryptocurrencies, smart contracts, and artificial intelligence. *AI Matters 1*, 2 (Dec. 2014), 19–21.

[53] Parameswaran, Sri, and Wolf, Tilman. Embedded systems security – an overview. *Design Automation for Embedded Systems 12*, 3 (Sept. 2008), 173–183.

[54] Poon, Joseph, and Dryja, Thaddeus. The bitcoin lightning network: Scalable off-chain instant payments. *Technical Report (draft)* (2015).

[55] Poon, Kara K. W., Yan, Andy, and Wilton, Steven J. E. A flexible power model for FPGAs. In *Proc. of the 12th International Conference on Field-Programmable Logic and Applications* (Montpellier, France, Sept. 2002), pp. 312–321.

[56] Pouraghily, Arman, Islam, Md Nazmul, Kundu, Sandip, and Wolf, Tilman. Privacy in blockchain-enabled iot devices. In *Internet-of-Things Design and Implementation (IoTDI), 2018 IEEE/ACM Third International Conference on* (2018), IEEE, pp. 292–293.

[57] Pouraghily, Arman, Wolf, Tilman, and Tessier, Russell. Hardware support for embedded operating system security. In *Proc. of the 28th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (Seattle, WA, July 2017), pp. 61–66.

[58] Provelengios, George, Pouraghily, Arman, Tessier, Russell, and Wolf, Tilman. A hardware monitor to protect Linux system calls. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (Hong Kong, China, July 2018), pp. 551–556.

[59] Provelengios, George, Pouraghily, Arman, Tessier, Russell, and Wolf, Tilman. A hardware monitor to protect linux system calls. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2018), IEEE, pp. 551–556.

[60] Rajkumar, Raj, Lee, Insup, Sha, Lui, and Stankovic, John. Cyber-physical systems: the next computing revolution. In *Proc. of the 47th Design Automation Conference (DAC)* (Anaheim, California, June 2010), pp. 731–736.

[61] Raspberry. Raspberry pi zero. https://www.raspberrypi.org/products/raspberry-pi-zero/.

[62] Roesch, Martin. Snort - lightweight intrusion detection for networks. In *Proc. of the 13th USENIX Conference on System Administration (LISA)* (Seattle, WA, Nov. 1999), pp. 229–238.

[63] Sanger, David E., and Perlroth, Nicole. A new era of internet attacks powered by everyday devices. *The New York Times* (Oct. 2016).

[64] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. of the IEEE Symposium on Security and Privacy* (Oakland, CA, May 2001), pp. 144–155.

[65] Sha, Lui, Gopalakrishnan, Sathish, Liu, Xue, and Wang, Qixin. Cyber-physical systems: A new frontier. In *Machine Learning in Cyber Trust*, Philip S. Yu and Jeffrey J. P. Tsai, Eds. Springer US, 2009, ch. 1, pp. 3–13.

[66] Sikiligiri, Amjad Basha. Buffer overflow attack and prevention for embedded systems. Master's thesis, Department of Electrical and Computer Engineering, University of Cincinnati, 2011.

[67] Stankovic, John A., Lee, Insup, Mok, Aloysius, and Rajkumar, Raj. Opportunities and obligations for physical computing systems. *Computer 38*, 11 (Nov. 2005), 23–31.

[68] Stritter, Edward, and Gunter, Tom. Microsystems a microprocessor architecture for a changing world: The motorola 68000. *Computer*, 2 (1979), 43–52.

[69] Suh, G. Edward, Lee, Jae W., Zhang, David, and Devadas, Srinivas. Secure program execution via dynamic information flow tracking. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Dec. 2004), pp. 85–96.

[70] Suh, G. Edward, Lee, Jae W., Zhang, David, and Devadas, Srinivas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (Boston, MA, Oct. 2004), pp. 85–96.

[71] Szabo, Nick. Smart contracts: Building blocks for digital markets, 1996.

[72] Theiling, Henrik. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on* (2000), IEEE, pp. 23–30.

[73] Thomas, Tedy, Pouraghily, Arman, Hu, Kekai, Tessier, Russell, and Wolf, Tilman. Multi-task support for security-enabled embedded processors. In *Proc. of 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (Toronto, ON, July 2015), pp. 136–143.

[74] uRaiden. https://raiden.network/micro.html.

[75] Vachharajani, Neil, Bridges, Matthew J., Chang, Jonathan, Rangan, Ram, Ottoni, Guilherme, Blome, Jason A., Reis, George A., Vachharajani, Manish, and August, David I. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. of 37th International Symposium on Microarchitecture* (Portland, OR, Dec. 2004), pp. 243–254.

[76] Waksman, Adam, and Sethumadhavan, Simha. Tamper evident microprocessors. In *Proc. of IEEE Symposium on Security and Privacy* (Oakland, CA, May 2010), pp. 173–188.

[77] Wiki, Bitcoin. Secp256k1. *Accessed: Aug 11* (2019).

[78] Wolf, Tilman, and Nagurney, Anna. A layered protocol architecture for scalable innovation and identification of network economic synergies in the internet of things. In *Proc. of the First IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI)* (Berlin, Germany, Apr. 2016), pp. 141–151.

[79] Wolf, Tilman, Zink, Michael, and Nagurney, Anna. The cyber-physical marketplace: A framework for large-scale horizontal integration in distributed cyber-physical systems. In *Proc. of the Third International Workshop on Cyber-Physical Networking Systems (CPNS) held in conjunction with the IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)* (Philadelphia, PA, July 2013), pp. 296–302.

[80] Wolf, Wayne. Cyber-physical systems. *Computer 42*, 3 (Mar. 2009), 88–89.

[81] Woodruff et al. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. of the International Symposium on Computer Architecture* (June 2014).

[82] Wörner, Dominic, and von Bomhard, Thomas. When your sensor earns money: Exchanging data for cash with bitcoin. In *Proc. of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UbiComp)* (Seattle, WA, Sept. 2014), pp. 295–298.

[83] Younan, Yves, Joosen, Wouter, and Piessens, Frank. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys 44*, 3 (June 2012), 17:1–17:28.

[84] Zhai, Xiaojun, Appiah, Kofi, Ehsan, Shoaib, Howells, Gareth, Hu, Huosheng, Gu, Dongbing, and McDonald-Maier, Klaus D. Method for detecting abnormal program behavior on embedded devices. *IEEE Transactions on Information Forensics and Security 10*, 8 (Aug. 2015), 1692–1704.

[85] Zhang, Yu, and Wen, Jiangtao. The IoT electric business model: Using blockchain technology for the internet of things. *Peer-to-Peer Networking and Applications 10*, 4 (July 2017), 983–994.