Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Michael S. Kirkpatrick

Entitled
Trusted Enforcement of Contextual Access Control

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Elisa Bertino, Ph.D.	
Chair	
Mikhail Atallah, Ph.D.	
Ninghui Li, Ph.D.	
Dongyan Xu, Ph.D.	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Elisa Bertino, Ph.D.

Approved by: William J. Gorman, Ph.D. 06/04/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Trusted Enforcement of Contextual Access Control

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Michael S. Kirkpatrick

Printed Name and Signature of Candidate

06/04/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

TRUSTED ENFORCEMENT OF CONTEXTUAL ACCESS CONTROL

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Michael S. Kirkpatrick

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2011

Purdue University

West Lafayette, Indiana

This work is dedicated to my children, Owen and Emily.

May you have the courage and freedom to pursue your dreams.

ACKNOWLEDGMENTS

I am privileged to have worked with so many outstanding friends and colleagues at Purdue University. First and foremost, this work would not exist without the dedication and support of Professor Elisa Bertino; our collaborations have been interesting, challenging, and rewarding. I am also grateful for the wisdom and guidance of Professor Gene Spafford, my advisory committee, Professors Mikhail Atallah, Ninghui Li, and Dongyan Xu, and the rest of the faculty. I am thankful to have known the late Amy Ingram, whose smile and kind words were always welcome. I am very appreciative to have worked with and known my colleagues and fellow graduate students, including Ian Molloy, Kevin Steuer, Ashish Kundu, Aditi Gupta, Salmin Sultana, Nabeel Mohamed, Jacques Thomas, and Gabriel Ghinita. Lastly, especial thanks go to Sam Kerr, whose innumerable contributions have been vital to this work.

Outside of Purdue, I am grateful to a number of collaborators who have supported my work in a multitude of ways. To Professor Richard Enbody at Michigan State University, thank you for introducing me to virtual machines and the academic world of security research. To my Oak Ridge National Laboratory mentor and friend Dr. Frederick Sheldon, it has been a pleasure to work with you. At Sypris Solutions, I express my gratitude to Dr. Hal Aldridge for supporting my work, and I look forward to continuing our collaboration.

Finally and most importantly, thank you to my family. To my wife, Brianne, words cannot express my gratitude; I could not have completed this journey without your unfailing support and unconditional love. To my children, Owen and Emily, I am indebted for your smiles that brighten any day, your hugs that melt my heart, and your giggles that bring pure joy.

PREFACE

My initial foray into the realm of computer and information security came as a software engineer for IBM Microelectronics Division (now Server & Technology Group). At the time, I was working on a team that implemented shape-processing algorithms for semiconductor manufacturing; these algorithms modified the chip designs to pre-correct microscopic errors that were introduced during the manufacturing process. This work started with three people as a feature that was nice to have. After about 10 years, it became a vital prerequisite for manufacturing, with more than 75 people working on the project.

My task was to design an access control system that restricted access to subsets of our code based on the person's job duties. Our team included infrastructure programmers, algorithm programmers, model designers, contractors, and project managers. Our computing environment included a mixture of Linux, AIX, and Windows workstations and servers, using a combination of multiple file systems. To spice the design up a bit more, we had to ensure compliance with corporate alliance partnerships, as well as federal legislation like international trade-in-arms (ITAR) regulations. I did not know it at the time, but I was implementing my first role-based access control (RBAC) project.

In the intervening decade between that project and this dissertation, I have implemented a variety of access control mechanisms using a diverse set of technologies. Although security involves a great range of interesting topics, it seems that I always return to fascinating topic of access control. With each new technology, there is always a new dimension that I have not explored previously; each project reveals a subtle difference from the last. Whenever I think I have mastered the topic, I find some new insight that I have hitherto missed. As the rest of my career unfolds, I would venture to guess that this field will continue to surprise and pique my intellectual curiosity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

AES	advanced encryption standard
APDU	application protocol data unit
API	application programming interface
ASIC	application-specific integrated circuit
ASLR	address space layout randomization
CCID	circuit card interface device
CDAC	context-dependent access control
CERT	computer emergency readiness team
CHAP	challenge-handshake authentication protocol
COTS	commercial off-the-shelf
CPU	central processing unit
CR3	control register 3
DAC	discretionary access control
DFC	digital fountain code
DNS	domain name service
DTE	domain-type enforcement
ELF	executable linkage format
FPGA	field-programmable gate array
GB	gigabyte
GEO-RBAC	geographical role-based access control
GHz	gigahertz
GIS	geographical information systems
GML	geography markup language
GOP	gadget-oriented programming

GOT	global offset table
GPS	global positioning system
HD	hard drive
I^2TD	intelligent insider threat detection
IND-CPA	indistinguishability against chosen plaintext attacks
IP	internet protocol
JSR	java specification report
KB	kilobyte
LBS	location-based service
LT	Luby transform
MAC	mandatory access control
MD5	message digest 5
MHz	megahertz
MLS	multi-level security
MUX	multiplexor
NDEF	NFC data exchange format
NFC	near-field communication
OS	operating system
OT	oblivious transfer
PD	page directory
PDA	personal digital assistant
PDP	policy decision point
PEI	policy, enforcement, and implementation
PEP	policy enforcement point
PIP	policy information point
PIR	private information retrieval
PKI	public key infrastructure
PLT	procedure linkage table
PPT	probabilistic polynomial-time

PT	page table
PUF	physically unclonable function
RBAC	role-based access control
RFID	radio frequency identifier
RISC	reduced instruction set computer
RO	ring oscillator
ROC	recovery-oriented computing
ROK	read-once key
ROP	return-oriented programming
RS	Reed-Solomon code
RSA	Rabin, Shamir, and Adelman
SDK	software development kit
SHA	secure hash function
SRAM	static random access memory
SSH	secure shell
SSL	secure sockets layer
TLS	transport layer security
TPM	trusted platform module
UCON$_{ABC}$	usage control – authorizations, obligations, and conditions
USB	universal serial bus
VMM	virtual machine monitor
VPN	virtual private network
WWW	world-wide web
XACML	extensible access control markup language
XML	extensible markup language
ZKPK	zero-knowledge proof of knowledge

ABSTRACT

Kirkpatrick, Michael S. Ph.D., Purdue University, August 2011. Trusted Enforcement of Contextual Access Control. Major Professor: Elisa Bertino.

As computing environments become both mobile and pervasive, the need for robust and flexible access control systems comes to the fore. Instead of relying simply on identity-based mechanisms or multi-level classifications, modern information systems must incorporate contextual factors into the access control decision. Examples of these factors include the user's location at the time of the request, the unique instance of the hardware device, and the history of previous accesses.

Designing and implementing such contextual access control mechanisms requires addressing a number of interesting challenges. First, one must be able to determine when the required policy conditions are satisfied. For instance, in the realm of spatially aware access control, the system must be able to validate user's claims to a particular location at a given time. Next, contextual mechanisms must be able to detect and react to changes in the environmental conditions, such as when a connection becomes disrupted. Finally, the integrity of the execution environment must be ensured, despite the complexity of modern computing systems.

To address these challenges, we have examined the creation of trusted enforcement mechanisms that are built on a combination of secure hardware, cryptographic protocols, virtual machine monitors, and randomized execution environments. We have developed a number of prototypes using NFC, PUFs, VMMs, and a microkernel OS to demonstrate the feasibility of our approaches to a number of contextual settings. Our experimental evaluation and security analyses demonstrate that robust mechanisms can be deployed for a minimal amount of computational expense.

1 INTRODUCTION

The history of computing has been marked by a trend toward smaller, more compact devices. Long gone are the days of Colossus and ENIAC, each of which occupied rooms. Mainframes gave way to minicomputers, which were surpassed by personal computers. Now, laptops, cell phones, and other portable devices dominate the marketplace of user devices.

As the landscape of computing changed, the field of access control evolved to reflect the new realities. When computing required the user's physical presence at the mainframe, restricting access could be accomplished by monitoring and controlling who could enter the room. Over time, it became necessary to partition users' permissions to different resources within the same computer. This need led to the development of such models as DAC, MAC, and DTE. As the complexity of identity-based access control grew, RBAC became commonplace as a method to reduce the maintenance burden for large organizations.

The computing industry is, once again, facing a paradigm shift in the field of access control. In the past decade, portable devices have become powerful and omnipresent. Corporate executives use their cell phones to send and receive emails. Medical workers carry tablet PCs containing patient records from one room to another. System administrators are assigned laptops to monitor remote systems from home. The diversity of computing environments creates a vast heterogeneity in the security assumptions that can be made for these systems.

In the case of email on smartphones, users frequently have the device store their passwords to ensure quick and easy access. As such, a thief can get quick access to vital company documents. Hospital laptops contain sensitive data. To ensure patient confidentiality, it would be desirable to ensure that this information could only be accessed from authorized locations. Public wireless access points may be compromised, and a malicious compromise could threaten the integrity of servers by corrupting remote administration requests.

While existing protection schemes offer a base layer of security for information systems, there is growing interest in enhancing these protections by examining the *context* of a request. For instance, a resource manager that can consider the user's physical location, the device being used, or the integrity of the execution environment may be capable of providing more robust security guarantees than one capable of only validating the credentials presented. In the former case, anomalies in these additional characteristics may indicate the presence of an attack. We describe this new approach as contextually-dependent access control (CDAC).

Study in traditional access control schemes has emphasized a separation between policy and implementation. This split was acceptable, as mapping the policy to the implementation was fairly straightforward. In the case of a DAC-based file system, a simple access control matrix could sufficiently express the way that file (object) permissions were granted to users (subjects). This simple construct was also sufficient for more complex systems, such as RBAC or DTE. These systems were not fundamentally different, but made administration easier by creating hierarchical definitions of subjects and objects. Formally, one could capture the essence of an access control policy with the following partial function:

$$Policy : Subject \times Object \to Permission$$

In contrast, CDAC introduces a new dimension to the domain of this function. To complicate matters further, this new dimension is inherently vague and is determined by the application domain. Consequently, the increasing gap between policy and implementation makes this dichotomy insufficient for CDAC. Rather, we adopt the view of Sandhu *et al.* [1] to consider PEI models (policy, enforcement, and implementation). In the PEI framework, the enforcement layer consists of architectures, protocols, and technological considerations that form a bridge between formal policies and the pseudocode of implementation models.

Up to this point, the primary focus of work in CDAC research has been the establishing the formalisms to reason about policies. Defining the architectures for enforcing these policies has largely been neglected in the research community. However, our work has demonstrated that addressing the challenges that arise from the enforcement of CDAC policies is worthy of study in its own right.

The theme of this work, then, is to examine how to combine cryptographic protocols, hardware technologies, and software techniques to create a root of trust for enforcing CDAC policies. Our work has been to explore these questions in a number of settings for various contextual factors. In each realm, we have also developed prototypes for empirically analyzing the merits of these techniques. In short, the intent of this dissertation is to document the feasibility of designing trusted enforcement mechanisms for CDAC.

1.1 Combining Location Constraints with RBAC

The first setting for our examination of CDAC is to incorporate location constraints into RBAC. RBAC is widely used in modern enterprise systems, as it eases the burden of administration by crafting policies based on roles, rather than identities. The increasing usage of mobile devices in enterprise settings, though, presents a new challenge as users need access to protected resources from a variety of settings. As such, it would be desirable to make a distinction between a VPN established through a public (and potentially malicious) wireless hotspot and one intiated from a secured office network.

Prior work on the topic of spatially constrained RBAC focused on augmenting policy models to incorporate logical or physical location data. However, existing work left a number of interesting challenges open for further consideration. For instance, the design of reusable enforcement architectures and protocols has received little attention. Addressing this challenge entails overcoming two hurdles. First, the system must provide a secure means to authenticate the user's claim to a particular location.

Second, as users are assumed to be mobile, the system must be able to enforce access control as the user's location changes.

To process a user's claim to a location, it would be inappropriate for an access control system to rely on something like GPS coordinates. GPS does not offer a way to determine the veracity of the user's claim. One could deploy a system where each device signs the coordinates, but this design still requires trusting the device. Similar to GPS, triangulation techniques in cell phone towers can be used to approximate the user's location, but this again requires trusting the phone. A more desirable approach would involve a protocol where the user retrieves a proof of location from a device in a fixed location. To ensure the proof is correct, guaranteed proximity to the location device is required. We explore the use of near-field communication (NFC) as a means to solve this problem.

After addressing the enforcement challenge, we then return to the policy level to introduce a new approach to spatially aware RBAC. In existing models, policies are based on the location of the requesting user. While this approach is the most intuitive, it is not the only paradigm for considering location information. In some settings, it is not the user's absolute location that matters, but the position *relative to other users*. Our proposed model extension and policy language allow the definition of *proximity constraints*, where the system considers the location of other users during the access decision.

We conclude our study of location constraints by inverting the threat model. Specifically, previous work in spatially aware RBAC assumes a one-sided adversarial question, where the system is designed to protect sensitive assets from potentially malicious users. However, when one considers the reality of malware and insider threats, it becomes apparent that the system–especially for the goal of *trusted enforcement*–must protect the user, as well. Specifically, an access control mechanism that can evaluate the contextual policies without revealing the user's location or identity would be very advantageous for preventing information leaks. Our work in this chapter provides a formal proof that it is possible to reconcile security with privacy in this regard.

1.2 Physical Contextual Factors

Although location can be an important factor in contextual access control, it is often desirable to consider the physical device itself. For example, consider a network of embedded devices responsible for monitoring and controlling aspects of a power plant or water treatment facility. Each such device may be assigned to a particular domain, and a supervisory system needs to enforce restrictions based on the device's identity.

In the second section of this dissertation, we adopt the challenge of controlling access based on the physical device being used. In our approach, we do not rely on any transient property or persistent cryptographic key. Rather, we leverage a hardware technology known as physically unclonable functions (PUFs) to authenticate a device remotely and securely. We then use the PUF to generate a one-time use symmetric key for protecting the data transmitted during the access session.

We then take this idea of PUF-based key generation a step farther. Specifically, we tackle the problem of dynamically generating cryptographic keys in a manner that they can only be used once (or a limited, configurable number of times). Although a naïve approach would involve writing the application software in such a way that the key is deleted, this scheme makes very strong implicit trust assumptions. For instance, the OS kernel could interfere, stopping the attempted deletion; a powerful adversary with physical access to memory could perform an attack that completely bypasses the software. To prevent these threats, we propose PUF ROKs (read-once keys) as a hardware design technique that incorporates the key generation and use into the processing unit. Consequently, the key never exists in a location that is susceptible to attack, and the hardware ensures that the key can only be used once.

1.3 Establishing a Resilient and Trusted Execution Environment

In our concluding section, we shift our focus from the policy and enforcement challenges to the execution environment itself. That is, applications cannot enforce complex policies with high assurance if an adversary has corrupted either the OS or the application code itself. Consequently, we end our discussion with techniques for establishing and preserving a trusted execution environment that is resilient against common and novel attack vectors.

We start this section by considering a very powerful adversary. We assume that a malicious actor has successfully corrupted the OS kernel in a highly targeted attack on a trusted application. The general consensus in the literature is that this adversary has "won the game," and all considerations of execution integrity are discarded. On the contrary, we explore techniques for integrating an authentication mechanism into a trusted virtual machine monitor (VMM) that executes at a privilege level below the OS. The aim of our technique is to detect *and repair* any damage to the application that occurs when the corrupted OS tampers with the application's memory image.

Although we have focused on the systems work of designing the recovery mechanism, this technique has interesting implications for CDAC. Specifically, one could couple this approach with attestation to validate the integrity of a remote system before granting access to a protected resource. When an attack occurs, the VMM would first attempt to repair the memory image; the access control decision would then be determined by whether or not the repair is successful and the application's integrity is ensured.

Finally, we turn to a weaker but more realistic adversarial model for application corruption. Specifically, we conclude by examining the threat of library-based attacks, primarily return-into-*libc* and what we call gadget-oriented programming (GOP) attacks[1]. While memory image randomization has been proposed as a means to stop

[1]The latter class has mostly been studied as return-oriented programming (ROP), because the attack is based on small pieces of code (gadgets) that end in `ret` instructions. However, more recent work has shown that gadgets can be constructed from other instructions, such as `jmp` and `call`. As such, the term ROP no longer fully captures the essence of the threat class.

Table 1.1
Concerns and technological basis for enforcement various CDAC constraints

Contextual Factor	**Primary Concerns**	**Technology**
Spatial awareness	Authentication of location, continuity of usage, relative proximity, location privacy	NFC, cryptography (PIR, OT)
Physical context	Distinguish "identical" devices, prevent leakage or modeling of device properties, avoid invasive threats	PUFs, ZKPK, FPGA
Execution integrity	Protection from corrupted OS, probabilistic automated recovery, buffer overflows	VMM, DFC, randomization, microkernel OS

these attacks, existing solutions suffer from a lack of run-time diversity. Our discussion, then, explores a new technique for strengthening defenses against this type of software vulnerability.

1.4 Summary and Document Structure

Table 1.1 identifies the primary concerns of each contextual factor under consideration, as well as the technological basis for our implementations. In the case of spatial awareness, it is crucial to validate the user's claim to a particular location, and to react accordingly as the user moves. Our prototype combines NFC cell phone technology, which has a tight proximity constraint, with cryptographic protocols to achieve our policy goals. In addition, we have extended the basic spatially aware RBAC model to consider other users' location and to protect individuals' privacy.

In considering the physical contextual factors, we have combined PUFs with zero-knowledge proofs of knowledge to distinguish physical instances of the same hardware device, while mitigating the threat of an attacker emulating the PUF through a model. We have also demonstrated a technique for using the PUF to generate one-time use cryptographic keys. Finally, we have proposed a VMM-based mechanism to protect the integrity of a trusted application's memory image, as well as a randomized process loader that protects the application against known buffer overflow exploit techniques. Coupling the former technique with attestation could allow one to examine *and repair* a remote application before approving its access request.

The remainder of this work explores these topics in detail. The outline can be summarized as follows. We begin with a state-of-the-art summary of related work in Chapter 2. We highlight work that forms the basis of our research, as well as other approaches with similar aims as ours. In Chapters 3, 4, and 5, we examine the enforcement challenges for spatially aware RBAC, including authentication of the user's location and credentials, defining new policy constraints, and protecting the user's privacy. Chapters 6 and 7 consider different contextual factors. In the former, we base policy evaluation on the device being used; the latter considers the history of accesses, by restricting the number of times that a cryptographic key can be used. Finally, our work in Chapters 8 and 9 considers the difficulty of ensuring proper execution of the applications that support CDAC, and we conclude in Chapter 10.

2 SURVEY OF RELATED WORK AND BACKGROUND MATERIAL

"If I have seen further than others, it is by standing on the shoulders of giants." This statement, often attributed to Isaac Newton[1], captures a fundamental aspect of any doctoral thesis. To understand a topic in detail, one must explore the existing literature on the subject. Here, we summarize the state-of-the-art relevant to CDAC.

2.1 Contextual Access Control Models

The first challenge in CDAC is to define precisely what is considered context. Intuition dictates that context should reflect the user's environmental conditions. Clearly, the user's physical location, represented by GPS coordinates, could be considered an example of context. In some settings, the logical location may be more useful; that is, the user's location is described in relative terms, such as "on the third floor," "in the hospital emergency room," or "in room 217." The precise GPS coordinates may be unnecessary for access control in such settings. As part of our work focuses on location as a special case to study in detail, we will examine the models for location-based CDAC in the next section.

Another aspect of contextual information for consideration is the trustworthiness of the principal. In order to quantify this factor, researchers have focused on the calculation of either risk or trust [3–6]. One challenge in this field is to design the system to adapt to new information. For example, many risk-based approaches involve defining weighting factors for pieces of data. However, it is not clear how a system should react when a user presents a new form of credential. Given the uncer-

[1]The aphorism actually predates Newton, as similar quotations have been traced to the Middle Ages. For an extensive history of the saying, see *On the Shoulders of Giants*, by Robert K. Merton [2].

tainty surrounding risk- and trust-based approaches, we do not focus on this direction as a form of CDAC.

2.2 Usage Control, PEI, and XACML

Our work on spatially aware RBAC was strongly influenced by previous work on usage control frameworks, PEI models, and XACML. The UCON_{ABC} family of models [7–9] describes the various methods for checking access requests. This framework describes conventional access control as *preA*, indicating the access check is performed before access is granted and is not performed again. In contrast, *onA* systems continue to enforce the access constraints while the resource is accessed. These continuous checks are important for mobile systems, where a user can move outside the permitted region after being granted access to a resource. We incorporate *onA* checks into the design of our architecture.

To bridge the gap between abstract policies and real implementations, Sandhu *et al.* have proposed the notion of PEI (policy, enforcement, implementation) models [1]. That is, the authors created a distinction between policy goals, which are traditionally high-level, abstract, and expressed in a formal manner, and enforcement mechanisms, which define the architecture, protocols, and technological constraints for creating an implementation. While GEO-RBAC describes the high-level policy, our work defines the enforcement model for deploying such a system.

XACML is an open standard for defining the structure of an access control enforcement mechanism [10]. One of the important elements of XACML is the separation of duties among multiple entities, including the policy decision point (PDP), the policy enforcement point (PEP), and the policy information point (PIP). The PIP is responsible for providing relevant information to the PDP in regard to a user's access request. Once the PDP has determined whether or not the access is granted, the decision is passed to the PEP, which is responsible for carrying out the decision. For example, the PEP may be a server that generates tickets that can be used to access

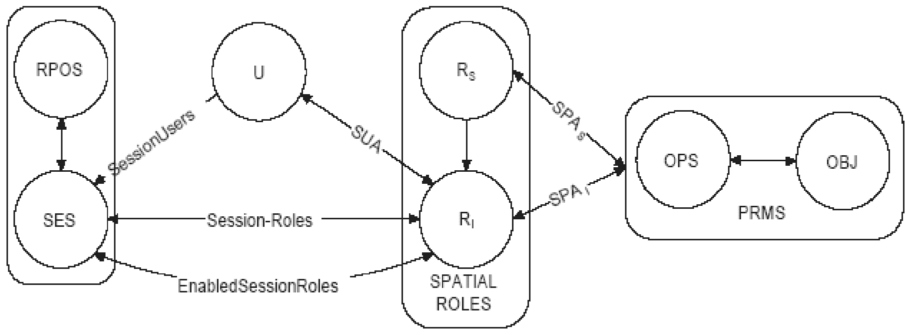


Figure 2.1. Core features of GEO-RBAC

the data. Part of our architectural work involves identifying the principals that make up the PIP, PDP, and PEP for a location-based RBAC system, as well as defining the protocols used to enforce the security guarantees.

2.3 Incorporating Location and Context into Role-Based Access Control

Role-based access control (RBAC) [11,12] is commonly used to model information system protections, including hierarchical designs [13, 14]. Several extensions to the basic RBAC model have been proposed, including some that incorporate temporal logic [15] and spatial constraints [16–21]. Additional work [22, 23] has focused on securing mobile and context-aware systems. These approaches have focused on abstract models to represent the spatial and temporal constraints, whereas our work focuses on creating an enforcement architecture and an implementation for such a system. In particular, our work expands on the GEO-RBAC model by examining the design necessary to enforce such constraints.

GEO-RBAC [18] introduces the concept of *spatial roles*, combining a traditional RBAC role with particular spatial extents. During a single session, a user is mapped to one or more spatial roles according to his or her location, as well as any credentials required to activate a role. *Permissions*, linking operations and objects that can be acted upon, are assigned to spatial roles. Thus, if a user can activate a particular spatial role during a session, he or she can then perform the actions specified by that role's permissions. The core notions of GEO-RBAC are shown in Figure 2.3.

There are two key novel features to GEO-RBAC. The first is the distinction between *role enabling* and *role activation*. When a user enters the region described by the spatial role's extents, we say that the role is *enabled*. However, the user cannot exercise any permissions associated with that role until he chooses to *activate* it. If the role is not activated, the user cannot exercise any of the associated permissions. The advantage of this distinction is that mutually exclusive roles can be defined for the same spatial region. Both roles can be simultaneously enabled, but only one can be activated at a time.

Another key feature of GEO-RBAC is the concept of *role schema*. A role schema is an abstraction, such as $<$ *Doctor*, *Hospital* $>$, that can be used as a template for singular roles. Permissions can be granted to role schemas in order to ease the administration of roles. A *role instance* is then created from a schema using specific data. For example, $<$ *Chief of Surgery*, *St. Vincent* $>$ can be an instance created from the $<$ *Doctor*, *Hospital* $>$ schema, as *Chief of Surgery* is a particular instance of *Doctor*, and *St. Vincent* is an instance of *Hospital*. Note that, while permissions can be granted to either schemas or instances, only instances can be activated. For additional details on GEO-RBAC, we refer the reader to the full paper.

A number of works have considered the enforcement challenges for spatial and contextual access control policies [24]. In Cricket [25], user devices analyze their own distance from known beacons within an indoor space; this approach assumes user devices make honest location claims in their access requests. The use of near-field communication (NFC) [26, 27] and RF-based sensors [28, 29] has received significant focus in the literature, while other works have considered techniques to ensure a user's contextual claim is consistent with those of other users in a mobile environment, such as a train [22]. Another approach includes the interposition of an access control engine *between* the user and the location service [30].

Our exploration of technologies to support a high level of integrity for location information has involved extensive use of NFC. NFC is an RFID-based proximity-constrained technology that provides contactless communication between a device and a reader/writer. However, NFC has a number of advantages over traditional RFID mechanisms, such as a very restricted broadcast range that is typically 10 cm in radius. This limited range is clearly sufficient to provide evidence of the user's presence in a room or building. Additionally, NFC defines a peer-to-peer mode that can be used to read and write data in a single contactless session. While recent work has uncovered attack vectors on NFC phones [31, 32], these attacks have focused on reading data stored for passive retrieval from an NDEF tag. Our design does not store such data, so these attacks are not related to our work. One work that is similar to ours is the approach developed by the Grey [27] project at Carnegie Mellon University. Grey is a smartphone-based system that is used to control access to secure rooms. In contrast to Grey, our aim is to incorporate NFC technology into an access control mechanism for information systems, not just to physical spaces.

Location is not the only factor commonly used as contextual information. Some models incorporate the time of the access request [17, 21, 33]. For example, an organization may wish to restrict access to sensitive data to business hours. Additionally, more subtle factors can be considered. The user's previous data access history may be required to enforce separation of duty or conflict of interest constraints. Environmental factors, such as the presence of a medical emergency or a criminal pursuit, may be sufficient to grant an access request that would otherwise be denied. Previous work in CDAC [34, 35] also includes aspects such as velocity or physical world conditions. Finally, context-awareness has also been applied to the unique challenges of ubiquitous computing in the home [22].

Our work on the privacy concerns of spatially aware RBAC is also related to the question of privacy-preserving queries in location-based services (LBS) [36] and k-

nearest neighbor queries [37]. However, as our work focuses on access control, the security concerns are more stringent. For instance, in most cases of LBS, the use of location is to provide local information to the user; if the user provides a location that is not accurate, he would receive inaccurate results. In spatial access control, however, the service provider must ensure that the user's location claim is correct. In addition, our model also employs RBAC, which requires authentication of the role, as well.

2.4 Physically Unclonable Functions and Device Identification

The literature of computer security contains a long history of identification schemes and authentication protocols [38–42]. Modern research in this area has become more focused on addressing issues concerning digital identity management under specialized circumstances, such as internet banking [43], secure roaming with ID metasystems [44], digital identity in federation systems [45], privacy-preservation for location-based services [46], and location-based encryption [47]. These works rely on knowledge or possession of a secret, and do not bind the authentication request to a particular piece of hardware.

The origin of PUFs can be traced to attempts to identify hardware devices by mismatches in their behavior [48]. The use of PUFs for generating or storing cryptographic keys has been proposed in a number of works [49–53]. The AEGIS secure processor [54] presents a new design for a RISC processor that incorporates a PUF for cryptographic operations. Our work contrasts with these, as we aim to integrate the unique PUF behavior directly into an authentication protocol, rather than simply providing secure key storage.

Relative to our work of applying PUFs to authentication and access control, [55] and [56] are perhaps the most similar. However, the former focuses on binding software in a virtual machine environment, whereas the latter focuses on authenticating banking transactions. Our protocols focus on light-weight multifactor authentication for distributed settings that require low-power solutions that minimize computation.

Other types of trusted hardware exist for various purposes. Secure coprocessors [57] and TPMs can provide secure key storage and remote attestation [58–61]. In many cases, the secure storage of TPMs can be used to bind authentication to a piece of hardware. However, we are interested in solutions for distributed computing that do not rely on TPMs, as TPMs may not be available for the devices used.

Finally, a new direction for hardware identification has emerged to identify unique characteristics of RFID devices [62–64]. These works are similar to previous work on PUFs, where they focus on identifying the device. These works do not propose new protocols that incorporate the unique behavior directly.

2.5 Virtual Machine Introspection and Integrity Enforcement

Authentication of executed code has received a lot of attention in recent years, due to novel applications such as outsourcing of services. Flicker [65] uses a trusted platform module (TPM) to prove to a remote party that a certain sequence of instructions is properly executed. An accumulated hash of a block of instructions is computed by a trusted hardware module, and the hash is signed with a key that is kept in trusted storage. In the Patagonix [66] system, the objective is to prevent execution of unauthorized applications. To that extent, the (trusted) OS maintains a database of hashes for the code and static data segments of known applications. Upon starting a new process, the OS checks if the executable's hash matches one of the values stored in the database, otherwise execution is prohibited. Note that, the setting in Patagonix is quite different from ours, since the OS is trusted. Furthermore, Patagonix does not address memory corruption that occurs dynamically at runtime.

Several solutions for secure code execution based on VMMs have been proposed. The Terra system [67] provides critical applications with their own virtual machine, including an application-specific software stack. However, such a solution may not be scalable, and may limit inter-operability. HookSafe [68] relies on a trusted VMM to guarantee the integrity of system call hooks that are used by applications. SecVisor [69] virtualizes hardware page protections to ensure the integrity of a commodity OS. XOMOS [70] proposes building an OS to rely on a trusted processor architecture to provide kernel integrity; the authors emulated the behavior in a trusted VMM, which could be used in place of the hardware. Similarly, SecureBit [71] proposes the use of external protected memory to defend against buffer overflows; like XOMOS, the authors emulated their design in a trusted VMM. These approaches either focus on protecting the integrity of the OS or make no attempt to recover after a memory corruption is detected. In contrast, our work focuses on creating an environment in which the application can be repaired and allowed to continue, even if the underlying OS is corrupt.

A number of other techniques have been proposed for resilient processing. For instance, checkpointing [72–74], returns the execution to a known, good state if a corruption occurs. Similarly, speculative execution [75, 76] performs a number of processing steps, and only commits to the results if they are correct. Some systems leverage multicore environments [77–79] and parallel execution [80] in an attempt to detect when an application has been corrupted. One such approach [81] will also resurrect a corrupted process automatically. CuPIDS [82] uses a physically separate co-processor to perform monitoring, similar to VMM-based approaches. Another interesting approach is to simply ignore faults that would lead to a crash and continue processing [83]. While all of these approaches have their merits, they do not consider the case where the OS itself is malicious.

Closest to our work on VMM-based integrity is the Overshadow [84] system, where a VMM prevents a malicious or compromised OS from accessing a protected application's memory space. The VMM encrypts the memory image of the application right after the corresponding process relinquishes the CPU, and before any other (untrusted) code gets the chance to execute. A hash with the digest of the legitimate memory contents is stored in trusted storage. However, Overshadow terminates the critical application as soon as corruption is detected. In contrast, we do attempt to recover the correct memory image and resume execution. Furthermore, the use of encryption in Overshadow leads to a more significant performance penalty than our approach. However, if encryption is vital to the protection of the application, our solution can be immediately extended to encrypt memory contents before encoding.

In another similar work, NICKLE [85] protects against kernel rootkits by keeping a duplicate image of kernel memory in a protected partition of the physical memory. Whenever an access to the kernel memory is performed, the running copy is compared to the saved copy, and an alarm flag is raised if a mismatch is detected. Similar to Overshadow and to our approach, NICKLE also relies on a trusted VMM. Note that, NICKLE only protects the kernel, but not applications running in user space, and the duplicate image must be created at bootstrap time. Furthermore, duplication of memory is not a scalable solution to protect multiple applications. To complicate things further, the memory image of applications changes frequently, whereas NICKLE only protects the kernel code segment which does not change.

Finally, ClearView [86] incorporates machine learning techniques into a VMM to patch running applications for common vulnerabilities without reboot. ClearView requires a learning phase, in which the VMM identifies correlated invariants, *i.e.*, actions that are associated with failures. When an invariant is violated, ClearView automatically generates a patch for the executing process. ClearView is aimed at untrusted applications that may be vulnerable to traditional exploits, such as buffer overflows; ClearView does not protect against corruptions that do not lead to application failures. In contrast, we offer a defensive mechanism to prevent external applications and an untrusted OS from corrupting any portion of a trusted application. That is, our approach can defend against corruptions that violate the integrity of the application data without leading to failure, whereas ClearView does not.

Our mechanism is reminiscent of previous attempts at the complex task of application recovery. Recovery-oriented computing (ROC), as proposed by Fox and Patterson [87], involves designing rapid failure recovery mechanisms into new and existing applications. The resulting programs include the ability to "undo" errors by returning to a good state. The approach that we adopt in this paper is to detect and automatically replace the corrupted application memory pages. As such, our work can be seen as a technique for transparently incorporating ROC principles into trusted applications.

Error-correcting codes (or erasure codes) have been extensively used in communication protocols for lossy channels. Maximum distance separable codes (e.g., Hamming codes) are optimal in terms of reception efficiency, but incur high encoding and decoding complexity (often quadratic to the size of the message). Such codes rely on parity checks, or polynomial oversampling. For instance, Reed-Solomon codes [88] employ oversampling of polynomials, and are used in broadband technologies (e.g., DSL), as well as Blu-ray discs.

Near-optimal erasure codes trade off some additional redundant storage requirements, but achieve fast computation time for encoding and decoding. Luby transform (LT) [89] and Raptor [90] codes fall in this category. We use LT codes in our implementation of the resilient execution environment for critical applications. Although Raptor codes are superior in principle to LT codes, they incur additional implementation complexity that may not be suitable for low-level deployment, such as within a VMM.

2.6 Return-into-*libc* and Return-oriented Programming Defenses

Address obfuscation [91] and address-space layout randomization (ASLR) (*e.g.,* PaX [92]) are two well-known techniques for defending against library-based attacks. Recall that one of the short-comings of ISR (described above) was that the small amount of randomization allowed brute-force attacks [93]. The same can be said for address obfuscation and ASLR on 32-bit architectures [94]. That is, Shacham *et al.* demonstrated that existing randomization techniques can be defeated by brute-force. While their proposed solution is to upgrade to a 64-bit architecture, thus increasing the number of possible randomized addresses, this approach has problems. First, for many settings (*e.g.,* embedded devices), upgrading to 64-bit architectures is simply not feasible. Second, and more problematically, information leakage can allow an attacker to learn the randomized base address of *libc* [95]. Consequently, simply randomizing the base address does not effectively block the attack.

Another approach for library-based attacks is to detect and terminate the attack as it occurs. DROP [96] is a binary monitor implemented as an extension to Valgrind [97]. DROP detects `ret` instructions and initiates a dynamic evaluation routine based on a statistical analysis of normal program behavior. When a `ret` instruction would end in an address in *libc*, DROP determines if the current execution routine exceeds a candidate gadget length threshold. These thresholds are based on a static analysis of normal program behavior. The binary to be run must be compiled with DROP enabled. DynIMA [98] combines the memory measurement capabilities of a TPM with dynamic taint analysis to monitor the integrity of the process in execution. Other approaches store sensitive data, such as return addresses, on a shadow stack and validate their integrity before use [99,100]. The disadvantage of these approaches is that there is a non-zero performance cost for every checked instruction. Also, with the exception of [100], these schemes assume gadgets end in `ret` instructions, and do not consider the more general gadget-oriented programming (*i.e.,* including gadgets ending in `jmp` or `call` instructions) case.

Compiler-based solutions [101,102] that create code without `ret` instructions have also been proposed. However, these techniques have the obvious disadvantage that they fail to prevent attacks based on `jmp` instructions. Compiler techniques have also been proposed to generate diversity within community of deployed code [103]. That is, instead of all users executing the same compiled image (*i.e.,* a monoculture), when a user downloads an application from an "app store" model, the compiler generates a unique executable, which would stop a single attack from succeeding on all users. While we find this approach very promising, it is not universally applicable, and would not stop an attacker with a singular target.

Perhaps most similar to our approach for stopping library-based attacks is that of proactive obfuscation [104]. This approach uses an obfuscating program that applies a semantics-preserving transformation to the protected server application. That is, the executable image differs each time the obfuscator runs, but the end result of the computation is identical. The *proactive* aspect means that the server is regularly taken off-line and replaced with a new obfuscated version, thus limiting the time during which a single exploit will work. Our work can be seen as another instance of proactive obfuscation. However, our focus is specifically on shared libraries in general applications, rather than long-running servers.

3 ENFORCING SPATIAL CONSTRAINTS FOR MOBILE RBAC SYSTEMS

Organizations have embraced role-based access control (RBAC) as a way to streamline the maintenance of access control policies. As a result of this popularity among enterprises, RBAC provides a very attractive foundation for incorporating contextual constraints into policies. Perhaps the most common extension is augmenting RBAC policies to reflect location constraints. With the increased adoption of mobile devices in the workplace, these constraints are becoming increasingly important. In this chapter, we will begin our exploration of location-based RBAC policies by considering the basic enforcement challenges. Specifically, we will propose a high-level framework for authenticating a request made from a mobile device, and we will describe the challenge of continuously enforcing the constraint as the user moves. This chapter also provides the basis for the subsequent discussions of novel location-based RBAC techniques.

3.1 On the Promise of Location-based RBAC

Location-constrained RBAC systems can offer robust fine-grained access control in a number of application scenarios. One example would be to improve the privacy of patient records in a health care system [105]. A limitation of current systems is the "bored but curious" employee; such a person may access the record of a celebrity undergoing treatment in the same hospital, despite having no valid reason to do so. While auditing provides a reactive security measure against such actions, a more proactive approach can establish a higher level of protection for user privacy. Incorporating spatial constraints could restrict access to the patient's record only to workers in the ward in which he is being treated, thus stopping the information leakage before it occurs.

In a government or military setting, secure processing of confidential material might require restricting such accesses to a single room or set of rooms. Simplistic, but undesirable, solutions could be to require a different set of credentials for use in the room or to restrict access to machines permanently stored in that location. A more flexible approach would be to permit users to bring in their (employer-assigned) mobile devices and present the same credentials they use otherwise. That is, enforcement of the spatial constraints would be transparent to the user.

To authenticate the user's location initially, we propose a novel proof-of-location framework, based on the assumption that a number of *location devices* are pre-deployed in known physical positions. A user retrieves the proof, including a timestamp, which he presents to a *resource manager*, along with other relevant credentials. The resource manager consults with a *role manager*, and grants a ticket for the resource if the request is approved.[1] We have also developed a prototype implementation of this protocol, where the user retrieves the proof-of-location using a cell phone equipped with NFC technology.

Our motivation for this work derives, in part, from the notion of PEI (policy, enforcement, implementation) models, as proposed by Sandhu *et al.* [1]. That is, the authors created a distinction between policy goals, which are traditionally high-level and abstract, and enforcement mechanisms, which define a re-usable structure for creating implementations. Consequently, we find that, while several location-based RBAC policy models exist, the enforcement question has received little attention in the literature. This chapter is intended to address the dearth of such architectures.

3.2 Supporting Policy Models for Location-based RBAC

Our solution is based on the GEO-RBAC [18] spatially aware RBAC model and incorporates elements of the UCON$_{ABC}$ family of access control models. The main feature of GEO-RBAC is the association of *spatial extents* with traditional RBAC

[1]Note that the *resource manager* and *role manager* are both servers, so the request decision is fully automated.

roles. GEO-RBAC policies can then use these *spatial roles* in defining fine-grained access control permissions. For instance, policies using the spatial role $< Manager$, *Room* 513 $>$ would require that the user activate the $Manager$ role (assuming the user is authorized) and be physically present in room 513. Consequently, a subject using the role $Manager$ would be denied access if the request is made from another location.

A key feature of GEO-RBAC is the differentiation of *role enabling* and *role activation*. A spatial role is automatically enabled if the user is authorized to activate the role and the user is physically present in the requisite location. However, an enabled role does not explicitly grant any privileges. Instead, the user must activate the role in order to exercise the associated permissions. This differentiation lets GEO-RBAC support complex policies, including mutually exclusive roles. It is the user's specific action that determines the role to be applied in a particular instance.

Another strength of GEO-RBAC is the support for hierarchical policy and role definitions. This support simplifies the administrative overhead of the model. For instance, an organization can define policies that grant permissions to <*Employee*,*Third Floor*>, where these permissions would simultaneously apply to a manager in Room 305 and a salesperson in Room 310, without requiring any redundant policies.

In this chapter, our interest in UCON$_{ABC}$ lies primarily with the notion of continuity of access. Specifically, UCON$_{ABC}$ policies include semantics that determine how the permissions are applied as the location or other conditions change. That is, as the user moves, the relevant policies may change and re-evaluation is necessary. As our consideration of UCON$_{ABC}$ in this chapter is very limited, we leave the discussion of this family of models for later chapters that rely on it more extensively.

3.3 Architecture

In this section we describe our architecture. We start by discussing the goals that shaped our design, then describe the assumed capabilities of the principals involved.

3.3.1 Design Goals

Our design approach was to keep our architecture as general as possible to provide for a diverse selection of implementations. In order to accomplish this approach, we defined the following goals for our design.

Maximize efficiency. The creators of the Grey smartphone-based access control system state, as a principle for designing security systems, "Perceived speed and convenience are critical to user satisfaction and acceptance." Consequently, any such access control protocol should be as efficient as possible. Our design aims to achieve to this goal by minimizing the number of communication steps and cryptographic operations for successful completion.

Separation of server duties. In a spatially aware RBAC system, there are two necessary steps to any access request. First, the requesting user[2] must be mapped to a role. Second, the role and request must be checked against the protected object's set of permissions. We model these distinct steps by designating separate principals for each.

Pseudonymize requests. When a user requests access to a resource, the server responsible for protecting the resource has no need for the user's identity information or the location. This server only needs to know what roles the user has activated. Only the server that maps the user to a role needs knowledge of the user's identity. We protect this data by encrypting it with a key known only to the user and the principal managing the role mappings.

Continuity of access. In mobile systems, a user could move outside the extents of the region for which a role has been defined. At that point, any request that was granted according to the user's original location should be revoked. We enforce this

[2]Note that there is some inherent ambiguity in relation to the term "user." We generally use the capitalized term *User* to refer to the physical device or the device software making the request, while the uncapitalized "user" typically refers to the actual person behind the request. In some cases, *User* may consist of multiple physical devices. For example, an NFC-enabled cell phone may be used for communication with the location device, while the actual access request is submitted to a resource after connecting the phone to a laptop. In such a design, *User* consists of the combination of the phone and the laptop.

constraint by using a continuity of access model that requires users to re-confirm their locations after a certain period of time. If the user has moved outside the allowed region, he will be unable to confirm his location, and his existing permissions will be revoked.

Generalized client design. In our design, we strive to make our system model as general and applicable as possible. That is, we desire to minimize any assumptions regarding the client's performance or security capacities. For example, we do not assume the user's mobile device is capable of multiple complex cryptographic operations, nor do we assume specialized hardware security mechanisms. Such assumptions would be barriers to adoption. Consequently, we cannot place any reliance on the trustworthiness of the client for determining the correct location. If the system were based on GPS, for instance, the server could not distinguish between a device that reported the true coordinates and a corrupted device that provided false locations.

3.3.2 Principals

From a high-level perspective, our design is based on a ticket-granting architecture in which a user submits an access request to the resource manager that owns the desired resource. If the request is granted, the manager issues a ticket the user can submit to the resource for the duration of the session. It is important to note that the resource itself is responsible for checking the validity of the ticket. However, as ticket validation is not directly related to the enforcement of location constraints, we consider such an issue to be outside the scope of this paper. Consequently, our architecture does not explicitly model the resource as a separate principal.

Although our discussion assumes a ticket-granting architecture, our design can also be applied when tickets are not involved. That is, if the system is set up so that the user issues an access request directly to the desired resource, then the resource itself is acting as its own manager. Once the access decision is made, the resource then grants access immediately without the additional step of issuing a ticket. However, for the simplicity of discussion, we will continue to refer to a ticket-granting design.

The following four principals form the core of our architecture.

- *User* – the principal making the request. This principal generally refers to the device used for the request, although one can also interpret it as the person making the request in some instances. When a distinction is necessary, we use the uncapitalized "user" to refer specifically to the person, whereas *User* would indicate the device.

- *Location Device (LD)* – the physical device storing location information. We assume that *LD* is installed in a pre-defined location and cannot be moved. For example, *LD* may be installed inside a wall or another immovable structure. *LD* serves as one part of the PIP, as it provides contextual information relevant to a request.[3]

- *Resource Manager (RsM)* – the resource manager responsible for the requested resource. *RsM* acts as both the PDP. If the policies regarding access to the resource grant permission based on *User's* currently active roles, then *RsM* approves the request and generates a ticket. As described previously, the resource itself (which is not modeled as a separate principal) acts as the PEP and takes responsibility for validating the ticket.

- *Role Manager (RoM)* – the role manager that maps a user to a set of roles. *RoM* is responsible for evaluating the location claim and the credentials presented. It then returns a list of active roles to *RsM*, which evaluates the request in relation to the defined policy. As such, *RoM* acts as the PIP. Although we assume *RoM* consists of a single, centralized server, we believe our architecture

[3]Our design differs slightly from the basic XACML structure in relation to the use of the PIP. Normally, the PIP is consulted by the PDP when it receives a request. However, this approach would require additional communication overhead, as *RoM* would have to contact *LD*, which could delay the access decision. Instead, *RoM* just needs to authenticate the data *User* gathered from *LD*. Consequently, our approach reduces the number of communication steps required.

could be applied to a distributed server, as well. We leave such a consideration for future work.

In practice, there would be multiple location devices and resource managers within the system. However, our protocol is designed to focus on a single access request at a time. In that view, *User* contacts a single *LD* for proof of location, then contacts a single *RsM* to request the access. Consequently, we only mention a single *RsM* and *LD* in our protocol definitions.

3.3.3 Communication

Figure 3.1 models the communication channels that exist in our architecture. With the exception of the channel between *LD* and *User*, we make no assumptions about the underlying network medium. That is, the other connections can be either wirless or wired, and we place no restrictions on this choice. However, as our design is based on the presumption *User* is mobile, we require the communication between *LD* and *User* to guarantee proximity.

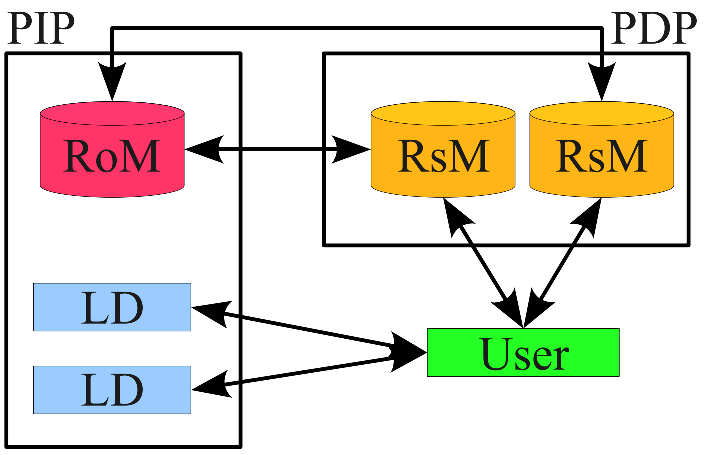


Figure 3.1. Communication channels within a spatially aware RBAC architecture

It is important to note that proximity is a relative concept, and the required precision would depend on the application. Our primary application scenario requires only loose constraints on location, such as *User's* presence in a particular room or suite. Consequently, absolute precision of the location is not required. Our design choice is to use the Near-Field Communication (NFC) technology that is available in certain Nokia cell phones. As previously stated, NFC's limited broadcast range of 10 cm is certainly adequate for ensuring that *User* is in the desired room. Furthermore, the reader could be placed inside a shielding device to block communication attempts from more powerful devices. Another advantage of this technology is that NFC supports a peer-to-peer mode that allows *User* and *LD* to exchange information simultanesouly. This capability is beneficial for the first two steps of our protocol.

3.3.4 Capability and Storage Requirements

As we previously described, we assume only a limited amount of computational power on the client-side principals, *User* and *LD*. Specifically, we assume that *LD* is able to perform a cryptographic hash algorithm, such as SHA-256. *User* must be able to perform symmetric key encryption, which entails the ability to store cryptographic keys securely.

On the server side, *RsM* must be able to perform symmetric key cryptography. *RoM* must be able to do the same, as well as perform the same cryptographic hash algorithm as *LD*. Additionally, *RoM* must be able to sign and verify certificates. As no other principal actually inspects the certificates used in our protocol, the signatures can be implemented using symmetric key cryptography. Doing so would improve the efficiency of the protocol. To simplify our implementation, we chose the symmetric key approach for the certificates.

In addition to these capabilities, each principal must store a limited amount of data. We summarize these storage requirements as follows.

Location device. Each device contains a certificate $Cert_{LD}$ signed by *RoM*. The certificate contains a unique identifier ID_{LD} and its physical coordinates, $Coords_{LD}$. The device also contains a password Pwd_{LD}.

User. Like *LD*, *User* stores a certificate $Cert_U$ that was signed by *RoM*. The certificate contains a unique identifier ID_U. *User* also has a password Pwd_U and shares a symmetric key K_U with *RoM* to encrypt its requests. In our design, K_U is stored in the secure element of the NFC cell phone, which means it can only be accessed by a trusted application. Additionally, *User* has a unique hardware identifier HW_U. The main purpose of HW_U is to bind the current request to *User*. As we will describe in Section 3.6, performing this binding is not a trivial feat. In the protocol, HW_U is revealed to *LD*, who binds the identifier to the current request as part of a cryptographic hash.

It is important to note that HW_U and K_U are associated with the physical device *User*. However, the certificate $Cert_U$ is associated with the person operating the device. If multiple people share the same device (for example, when nurses in a health care setting share a laptop), then the device would need a mechanism for switching certificates. Similarly, if the battery in one user's device is almost out of power, the user can use a Bluetooth connection or flash storage to transfer his $Cert_U$ to another device.

Resource manager. Each resource manager is responsible for controlling access to a set of resources by granting tickets to users. The manager stores its access control policy, $RolePerms$, that maps permissions for resources to roles. *RsM* also maintains a list, denoted by *CrntTix*, of valid tickets it has issued. Tickets are removed from this list when they become invalid, which can occur when the ticket expires, the user deactivates the role, the user activates a conflicting role, or the user moves out of the spatial extents of the region associated with the role. Recall that *RsM* does not have knowledge of *User*'s identity. Instead, *RsM* generates a unique session identifier ID_S for each access request, and associates the ID_S with the corresponding ticket granted.

Role manager. The role manager maintains the authoritative $RoleMap$, which maps users to roles and roles to geographic locations. For each *User*, *RoM* stores a list of $ActivatedRoles$. *RoM* also keeps a map, $UserResMap$, that associates *Users* with *RsMs* based on requests made. That is, $UserResMap$ stores pairs of the form $<$*RsM*,$ID_S$$>$ for each *User*. The utility of this map is evident when considering mutually exclusive roles. If *User* activates a role that conflicts with a previously activated role, *Role* must inform the appropriate *RsM* that the previous role has become de-activated. When a *RsM* determines that a session has ended, it sends a request to *RoM* to remove the relevant $<$*RsM*,$ID_S$$>$ entry from $UserResMap$.

RoM also maintains a number of cryptographic keys and tokens. *RoM* shares a unique symmetric key K_U with each *User*. The key is identified by the token HW_U. *RoM* also stores a key (or key pair) for signing and verifying certificates. As previously described, in our implementation, we used symmetric key encryption for the certificates, as no other principal needs to verify the certificate. Finally, *RoM* stores the set of passwords for each user and *LD*.

3.3.5 Setup Phase

Setting up an implementation of our protocol requires a number of steps. First, *RoM* must generate and sign the certificates for each *User* and *LD*. For *LD*, generation of the certificate and creation of the password Pwd_{LD} must occur *before* the device is installed in its physical location. Consequently, proper controls must be in place to prevent a malicious administrator from installing *LD* in a false location.

For *User*, the certificate generation occurs when the user registers with the system. The user can create an initial password when registering, and can change the password at any time. Next, HW_U and K_U must be generated and installed in the phone. As K_U must not be leaked, we install it in the phone's secure element, which restricts access to trusted, signed applications.

3.3.6 GEO-RBAC

As mentioned previously, our work is based on the GEO-RBAC model for spatially aware RBAC. Recall that GEO-RBAC makes the distinction between an *enabled role* and an *activated role*. A spatial role is automatically enabled once a qualified user enters the spatial extents. Role activation, however, is performed only in response to a specific request by the user. The role must first be enabled before the user can request its activation. He must also provide the requisite authorization credentials.

In our framework, when *User* submits an access request, he specifies the role to activate in order to satisfy the access control policy. To process the request, *RoM* must compute the appropriate set of activated roles. The algorithm in Figure 1 describes the calculation. First, the set of currently active roles is intersected with the currently enabled roles. That is, if *User* has left the spatial extents of a previously active role, it is no longer enabled and cannot be considered active. If the requested role is in this intersection, that means it is enabled and has previously been activated. If the role is not in the intersection, then it is added to the set of activated roles. However, as GEO-RBAC supports mutually exclusive roles, the algorithm must remove any conflicting roles that have previously been activated. The algorithm then returns the set of activated roles for the user.

Algorithm 1: ComputeActiveRoles

```
Input: $r$ : the activated role ; $l$ : the requesting user's location
Output: $R$ : the set of active roles
$R \leftarrow \textit{current active roles}$ ;
$E \leftarrow \emptyset$ ;
$S \leftarrow \textit{spatial roles}$ ;
foreach $s \in S$ do
    if $l$ inside $SpatialExtents(s)$ then
        $E \leftarrow E \cup \{s\}$ ;
$R \leftarrow R \cap E$;
if $r \notin R$ then
    $R \leftarrow R \cup \{r\} - ConflictingRoles(r)$;
return $R$ ;
```

3.4 Protocols

Our architecture requires multiple protocols for granting and maintaining access. In this section, we start by describing the protocol for making an initial request, and then present the protocol for maintaining continuity of access according to the $UCON_{ABC}$ model.

3.4.1 Initial Request

The initial access protocol, in which the user requests access to a resource, consists of the following steps. Graphical representation of this protocol is shown in Figure 3.4.1.

1. $[\mathit{User} \rightarrow LD : HW_U]$ The user's device sends its hardware identifier HW_U to the location device, which binds the proof of location to the requesting device.

2. $[LD \rightarrow \mathit{User} : \ Cert_{LD},\ T,\ H^*]$ Using the hardware identifier received in step 1, *LD* generates a timestamp T and computes a cryptographic hash $H^* = H(HW_U, Pwd_{LD}, T)$. This binds *User* to the current location at the time given by the timestamp.

3. $[\mathit{User} \rightarrow RsM : HW_U,\ T,\ E^*]$ *User* creates an encrypted package E^* containing the requested role to activate (*Role*), the hash H^* that provides proof-of-location, the user's password Pwd_U, and the two certificates signed previously by *RoM*. That is, $E^* = E_{K_U}(\mathit{Role}, H^*, Pwd_U, Cert_U, Cert_{LD})$. The encryption is performed with a symmetric key that is shared between *User* and *RoM*. The hardware identifier HW_U is sent in an unencrypted form to permit *RoM* to look up the corresponding key E_{K_U} for decryption.

4. $[RsM \rightarrow RoM : HW_U,\ T,\ E^*,\ ID_S]$ *RsM* forwards the HW_U and E^* packages received from *User*, along with the timestamp T. In addition, *RsM* generates a

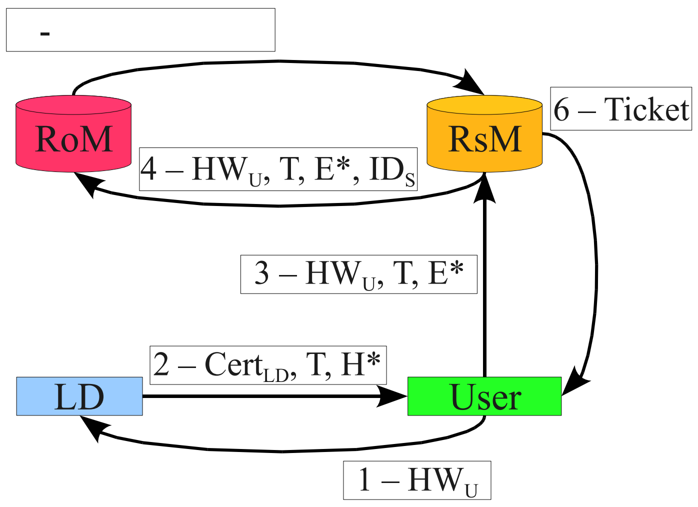


Figure 3.2. Data transferred at each step of our protocol

session identifier ID_S. This identifier allows *RoM* to create a mapping between *User* and the current request, but without revealing the actual request to *RoM*.

5. [$RoM \rightarrow RsM : ActivatedRoles$] After receiving the data from *RsM* in the previous step, *RoM* uses HW_U to look up the key associated with *User* and decrypts E^*. *RoM* then validates the certificates and checks the user's password Pwd_U. *RoM* then looks up the password for *LD*, and reconstructs H^* using HW_U and the given timestamp. If the hashes and the passwords match, *RoM* computes *ActivatedRoles* by executing Algorithm 1. The resulting list is stored and set to *RsM*.

6. [$RsM \rightarrow User : Ticket$] *RsM* examines the received *ActivatedRoles* list and applies the access control policies. If the policy is satisfied, *RsM* issues a ticket that can be used to access the requested resource.

There are a couple of key features to our initial protocol that may not be intuitive and require justification. First, consider the session identifier ID_S. While the use of this identifier may not be obvious, it is beneficial for mutually exclusive roles and continuity of usage. As such, we will describe its use in the following sections.

Next, the need for both steps 3 and 4 requires consideration. One could argue that it would be more efficient for *User* to send the request directly to *RoM*, rather than *RsM*. One problem with such an approach is that step 3 is sent *along with the request*. That is, step 3 involves additional information that we do not explicitly model, as it is dependent on the application scenario. The two steps help to preserve pseudonymity, as *RsM* has knowledge of the request, but not the requester's identity or location; *RoM*, on the other hand, is aware of the identity, but has no knowledge of the request being made.

An additional advantage of keeping steps 3 and 4 distinct is that it maintains the locality of policies in *RsM*. One resource manager may require strict adherence to a time frame and may reject any request where the timestamp is more than a couple of seconds old. Another may allow requests if the timestamp shows the user was at the location an hour ago. Clearly the latter creates a much looser interpretation of the location constraint (i.e., the user only needs to prove that he *was* in that location, not that he still is), but our approach allows the administrators of the resource managers the flexibility to implement such a policy.

More subtly, the preservation of steps 3 and 4 is a defensive mechanism against a denial-of-service attack. Recall that we assume a single, centralized *RoM*, but several *RsMs* distributed throughout the network. If a compromised *User* sends a large number of requests to *RoM*, that server could become overloaded and the entire system could crash. If a number of the *RsMs* are enabled with mechanisms to detect the attack, they could prevent the attack on *RoM*.

A second possible criticism of our protocol is the lack of cryptography protecting the messages between principals. This choice is deliberate, as our protocol is intended as an overlay on top of an existing network infrastructure. That is, we assume that system implementers apply cryptographic techniques as necessary. Cryptography could be used in step 5 to prevent an attacker from modifying the *ActivatedRoles* in transit. However, in some applications, *RoM* and *RsM* may exist on the same physical machine, so encryption may be unnecessary. Thus, we only explicitly model the cryptographic mechanisms required to achieve our stated goals. In our implementation, *RoM* and *RsM* exist on the same machine, so encrypting steps 4 and 5 was unnecessary.

We also observe that steps 1 and 2 depend on the choice of proximity-enforcing technology. That is, some implementation choices would result in step 1 being unnecessary, while others would modify the hash value H^*. Our implementation choice is to use the peer-to-peer mode available in the Nokia NFC technology. This mode allows *User* to send HW_U to LD and receive the resulting data all within a single contactless connection. For other technology choices, designers may have to modify these steps as appropriate.

3.4.2 Mutually Exclusive Roles

Our base protocol, as described above, does not fully support mutually exclusive roles. While this may be appropriate for some applications, there are others that require such support. Assume that $ActivatedRoles_n$ denotes *User's* current set of activated roles. For a new request, *RoM* computes the new set $ActivatedRoles_{n+1}$. If $ActivatedRoles_n \subset ActivatedRoles_{n+1}$, *User* has activated a previously unused role. In some settings, broadcasting $ActivatedRoles_{n+1}$ may be required to ensure this new role does not violate the policy of some *RsM*, but this message is unnecessary in general. However, if $ActivatedRoles_n \not\subset ActivatedRoles_{n+1}$, the newly activated role has forced the disabling of a previously used role. As a result, whatever permissions *User* is exercising as a result of the previous role should be revoked. This revocation is enabled by ID_S.

Recall that *RoM* maintains the data structure $UserResMap$ for each *User*. This structure consists of pairs of the form $<RsM, ID_S>$. *RoM* can use these entries to notify the relevant *RsM* of the updated $ActivatedRoles_{n+1}$. Specifically, we introduce two optional steps to our protocol.

4a. [$RoM \rightarrow RsM^*$: $ActivatedRoles_{n+1}$] *RoM* sends a broadcast to the list *RsM** consisting of the resource managers with which *User* currently has an active session (according to $UserResMap$). This message lets these resource managers revoke any outstanding tickets, if necessary, according to the policies maintained locally by the *RsMs*.

4b. [$RsM^* \rightarrow RoM$: Ack] This optional step can be used to enforce strict mutual exclusion by requiring that the new request can only be approved after the *RsMs* have had a chance to revoke the necessary tickets. If strict requirements are not needed, this step can be omitted.

3.4.3 Continuity of Access

In terms of the UCON$_{ABC}$ model, our basic protocol defines a *preA* approach to access control. This means that the permissions and credentials are checked only before the access is granted. In systems where users are mobile, one could move out of the extents of the spatial role after being granted access to a resource. As a result, we would like to enforce *onA* constraints, as well. These constraints require *User* to confirm his position while accessing the resource.

User can be required to confirm his location in either a proactive or reactive manner. As *RsM* serves as the PDP, it would control which method is applied. In the proactive approach, *User* would initiate the confirmation protocol; to reduce the reliance on a person's memory, the client software would display a reminder. In the reactive approach, *RsM* would connect with *User* and request confirmation. In either approach, the burden on the user should be minimized.

Once confirmation is required, the previous protocol is modified as follows.

1. [$User \rightarrow LD$: HW_U] This step works as before.

2. [$LD \rightarrow User$: $Cert_{LD}$, T, H^*] This step works as before.

3. [$User \rightarrow RsM : T,\ E^*$] The hardware identifier HW_U is unnecessary, as *RsM* has stored this value for the session. In most cases, the intent is simply to re-establish *User's* location, the role and *User's* credentials are not needed. That is, $E^* = E_{K_U}(H^*, Cert_{LD})$. However, in a more secure setting, the user may be required to re-enter his password, and E^* would be constructed as above.

4. [$RsM \rightarrow RoM : HW_U,\ T,\ E^*$] As *RsM* has stored HW_U, it adds the identifier to the message to *RoM*. Doing so prevents a collusion attack in which a different *User* device is used for a false confirmation. The ID_S is not needed at this point, so it is omitted.

5. [$RoM \rightarrow RsM : ActivatedRoles$] *RoM* computes the new set of *ActivatedRoles* and sends the updated list to *RsM*. As there is no new request to activate a role, the computation is simply checking whether the existing roles are still enabled according to the new location.

If the updated *ActivatedRoles* list continues to satisfy *RsM's* policy, the access is allowed to continue. Depending on the ticket-granting implementation, *RsM* may issue a new ticket, it may contact the resource directly and inform the resource of an updated expiration date for the ticket, or not action may be necessary. Similarly, if *User* is unable to confirm his location as required by the *RsM's* policy, *ActivatedRoles* would be null, and *RsM* could begin the process of ticket revocation. In either case, the action mechanics are external to the issue of location constraints. As such, we consider this topic to be beyond the scope of this paper and leave the question to implementation designers.

In considering continuity of access, step 5 of the protocol becomes much more complicated. In Section 3.4.2, we introduced the notion of broadcasting the updated *ActivatedRoles* list to other *RsMs* before responding to the current request. If *User* has successfully confirmed his location and all activated roles are still enabled, then no action is necessary. However, if the confirmation fails, or the *ActivatedRoles* has changed, a number of possibilities arise.

If the confirmation fails, a conservative approach would be to inform all *RsMs* so that they can provide an appropriate response. In general, though, we believe such an approach is undesirable. Consider the case where the user makes a mistake re-typing his password. Revoking all of this accesses would not be appropriate. Instead, a better response would be simply to let the current *RsM* handle the failure. Other *RsMs* would continue to enforce the continuity of usage according to their own policies, and there would be no overhead penalty of false-alarm messages.

Another possibility is that the confirmation succeeds, but a subset of *ActivatedRoles* have become disabled. That is, the user has confirmed both his location and his identity, so there is no chance of a false-alarm as above. In this case, *RoM* should broadcast the new list to the *RsMs* in the entries in $UserResMap$. The *RsMs* would then have the ability to adapt to the change in environment. Unlike the case of mutually exclusive roles, though, we do not see an advantage in delaying the confirmation step. Thus, after broadcasting the updated *ActivatedRoles*, *RoM* immediately informs the current *RsM* of the confirmed roles.

Note that in this continuity of access model, there is a delay between when *User* leaves the spatial role's extents and when the role is deactivated. In many cases, this would be acceptable. In high-security settings, this delay could be eliminated by having explicit entrances and exits for the spatial role. Then, the door can be treated as a resource and the role required for access is mutually exclusive with any role enabled by the location device within the spatial extents.

3.5 Implementation and Evaluation

We have developed a prototype implementation of our architecture using Java. To implement *RoM* and *RsM*, we have adapted the source code of the GISToolkit [106]. This library implements the OpenGIS Geography Markup Language (GML) encoding standard [107]. We have extended the library to define spatial regions appropriate for modeling a building. Specifically, our extension models floors, rooms, and suites.

We have also incorporated XML files to represent the role definitions, spatial role extents, and permissions.

On the client side, we have split the behavior of *User* into two pieces. The first piece consists of a Nokia 6131 NFC-enabled cell phone [108, 109]. We use this device to connect with an Advanced Card Systems (ACS) NFC reader, model ACR 122 [110], which serves as *LD*. The second piece of *User* is a Java client application written for a more traditional computing device, such as a laptop. The data generated by *LD* in step 2 of our request protocol is transferred from the phone to the laptop manually.

Implementation of *User* in these two pieces is problematic. First, it is inconvenient. More importantly, it breaks the guarantee that the user of the laptop is in the claimed location; colluding users could simply communicate the data from *LD* via some side communication channel. Both of these criticisms can be addressed by our vision of integrating NFC technology into a custom computing device.[4] Access to HW_U could then be controlled by trusted hardware, such as a Trusted Platform Module (TPM), ensuring that the request is bound to a known and trusted device.

Communication between *User*, *RsM*, and *RoM* is accomplished using traditional sockets. Implementing the communication between *User* and *LD* was more challenging. On the Nokia 6131 NFC phone, we deployed a MIDlet that transmitted data using the Java JSR 257 Contactless Communication API [111]. The ACR122 reader was connected via USB to a Windows XP workstation.

Implementing the behavior of *LD* required the development of a custom Java application to communicate with the reader. The ACR122 uses the standard Windows CCID interface. While the SDK provides documentation and sample code for traditional smart cards (e.g., only reading or only writing NDEF-formatted tags), this approach is undesirable for our design. That is, this basic approach would require connecting the phone to the reader twice; the first touch would send data from the

[4]While one may object to this solution, we argue that an organization requiring a spatially aware RBAC system is likely to have the resources available to design such a device, or contract to a company that will do so.

phone to reader, and the second touch would be the response. Furthermore, the user would have to intervene manually to change the modes between the touches.

A better approach is to use the peer-to-peer extension of JSR 257, which is supported by both the Nokia 6131 NFC and the ACR122. Although the SDK provided by ACS does not contain examples of this functionality, we were able to adapt code from the nfcip-java library [112]. Combining the APDUs (instructions for communicating with the ACR122) with the basic structure of the ACS SDK, we were able to implement the peer-to-peer communication.[5] The phone is designated as the initiator of the request (as it sends data first), while the reader is then designated as the target.

Two machines were involved in our performance evaluation. The first test machine was running Windows XP on an Intel Pentium M CPU running at 1.60 GHz with 1.3 GB of memory clocked at 333 MHz. We attached the ACR122 reader to this machine via a USB connection to measure the amount of delay experienced by the user as a result of the NFC communication. Initiating the peer-to-peer connection 50 times, we observed an average of 131.4 ms delay from time the request is sent from the phone until a response is received.

Our second machine was used to test the overhead of the back-end server portion of our design. This machine was running Ubuntu Linux, version 9.04 (Jaunty Jackalope) on an Intel Core 2 Duo CPU running at 2.26 GHz with 3 GB of memory clocked at 667 MHz. To eliminate network delay from our measurement, our implementations of *User*, *RsM*, and *RoM* were all executing on this machine, using sockets to communicate. We measured the delay from the time that *User* submitted the request to *RsM* until it received a response. On average, our protocol yielded an overhead of 24.4 ms. Clearly, the overhead imposed by the local computations of our architecture is minimal, and most of the delay users observe would result from normal network

[5]One may question why we did not use the nfcip-java library as written. We were unable to get this library working with our model of the reader. After contacting the library's author, we learned that the firmware of our model may be incompatible with the library. We are grateful to the author for his willingness to help with this problem.

communication. As such, we argue that implementing our design for a real spatially aware RBAC is certainly feasible.

3.6 Security Analysis

In this section, we present an informal security analysis that primarily focuses on two threat models. As our primary concern is to secure access to the protected resource, we mainly examine the threat from a malicious *User*. However, we also consider the threat posed by *RsM*, as one of our design goals is to ensure the pseudonymity of *User*. The aim of our security analysis was to address common attacks on authentication protocols [113–117]. These attacks include *replay*, *collusion*,*reflection*, *denial-of-service*, and *typing*. We do not consider *eavesdropping* and *modification*, as our system is built on the assumption that appropriate cryptographic mechanisms are used to protect messages at the lower layers of the network stack.

Assumptions. Given these threat vectors, we state a number of assumptions. First, we assume the integrity of *RoM*. As no other principal has knowledge of the mapping between users and spatial roles, we do not see a defense against a corrupted *RoM*. That is, if a corrupted *RoM* reports false *ActivatedRoles*, the attack may be detected, but there is no mechanism for correction. Consequently, we assume that the *ActivatedRoles* reported in step 5 is correct.

Additionally, we assume the coordinates stored in *LD's* certificate are correct. Note that our assumption does not preclude the chance that a malicious *User* can attempt to use a false certificate. Next, we assume that *RsM* is acting in good faith to protect the resource. If not, *RsM* could simply issue tickets without regard for the protocol.

Malicious *User*. In this attack, the primary goal of the attacker is to gain illicit access to the protected resource. A secondary goal would be a denial of service for others. Our threat model assumes the attacker is computationally bound, *i.e.*, he cannot break the cryptographic hash or encryption primities employed. Under this model, our protocol is secure against the following attacks as explained below.

1. **Replay** The goal of a replay attack is for *User* or an eavesdropper to reuse a piece of data as part of a false request. Clearly, the $Cert_{LD}$ and timestamp T are tied to the given request (and the *User* through the hash $H(HW_U, Pwd_{LD}, T)$. If the timestamp is modified or a different $Cert_{LD}$ is sent, then the hash would not match. Similarly, if the hardward identifier HW_U is changed in transit, again, the hash would not match. Furthermore, HW_U is tied to the given *User*, as the symmetric key is only shared between that *User* and *RoM*.

   Another advantage of the timestamp T is that it ensures the timely use of the location information. That is, *User* cannot hoard the data received from *LD* for use after moving out of the spatial extents. First, the data may be marked as invalid if T is beyond an acceptable time frame. Additionally, T can be used to enforce an ordering of the requests. That is, if $T_1 < T_2$, but T_2 arrives at *RoM* first, the request with T_1 would be denied as an expired request.

2. **Collusion** There two possibilities for collusion between two users. The first attack is for *User*$_1$ to obtain the proof-of-location from *LD*, and send the proof to *User*$_2$ via a side channel. This attack is essentially identical to *ghost-and-leech* attacks on RFID readers. As described in Section 3.5, our vision for deployment with a custom device would obviate this attack vector, as only known and trusted devices can submit requests to *RsM*. A second possibility for collusion would be for *User*$_1$ to send a valid ticket to *User*$_2$. Although we generally consider the implementation of the ticket-granting service to be beyond the scope of this paper, we believe that using remote attestation techniques with a TPM could address this threat.

3. **Reflection** In a reflection attack, the attacker would engage in a protocol with a target to get data that could be reused as part of a request. In our design,

the target would have to be another *User*, as no other principal reveals credentials that the attacker could attempt to reuse. However, in our protocol, *User* only initiates the protocol. That is, a malicious $User'$ cannot initiate the protocol with a targeted *User*. Thus, a reflection attack is not applicable in our architecture.

4. **Typing** For a successful typing attack to occur, there must be more than one piece of data of the same type. That is, the attacker tries to get the victim to accept one piece of data as another based on the two pieces having the same format. In our protocol, there are no instances of two pieces of data having the same type. As a result, a typing attack is not possible.

5. **Denial-of-service** Our protocol is designed so that all *Users* must submit their requests through a distributed number of *RsMs*. If *User* attempts a denial-of-service attack against a particular *RsM*, he may succeed depending on how robust the *RsM* is against such an attack. However, this attack vector is not the result of our design, but is an inherent danger of a networked system. *RsM* could mitigate the damage from these attacks by employing appropriate measures, such as blacklisting suspected nodes.

   RoM, as a centralized server, can also be a target for a denial-of-service. However, our assumption is that the *RsMs* are behaving properly. As a result, *RoM* could send a request to the *RsMs* to throttle their requests as appropriate. Furthermore, if *RoM* is equipped with software to analyze recent logged requests, it could identify a potentially misbehaving *User* by identifying any HW_U identifiers that are associated with an abnormal number of requests. Thus, while both of these types of denial-of-service are possible, it is our observation that these attacks are not contingent on the design of our architecture, and a number of mitigation techniques are possible.

Malicious *RsM*. The goal of a malicious (or corrupted) *RsM* would be to bind the user's identity to the requests. However, the encryption of the $Cert_U$ ensures that

RsM is restricted to only the pseudonymized identifier HW_U. That is, if the attacker discovers which user had possession of a particular *User* device at a given time, he could discover the identity by pairing the user with the associated HW_U. Although this is a breach of strict confidentiality, our design goal was to provide pseudonymity, rather than pure anonymity. That is, without the ability to pair a user with the *User* device, the attacker cannot discover the identity through the request alone.

3.7 Conclusions

In this work, we have proposed a novel architecture for enforcing spatial constraints in an RBAC environment. We identified a number of goals that such an architecture should meet, and constructed protocols that accomplish these goals. We have demonstrated that our architecture is flexible and can be applied to a number of settings with varied security requirements by localizing the policies in the individual *RsMs*. Our design incorporates concepts from the $UCON_{ABC}$ usage control model to enforce continuous access checks while the user accesses the protected resource.

We have implemented a prototype of our architecture that provides a proof of concept. Our prototype uses a Nokia 6131 NFC cell phone to communicate with a ACR122 reader connected to a workstation. We have addressed the challenges we encountered in adopting this technology, and described the performance we observed in our experimental evaluation. We have also provided an informal analysis of the security guarantees of our design.

4 PROX-RBAC: A PROXIMITY-BASED SPATIALLY AWARE RBAC

In the previous chapter, we explored the question of how to enforce location constraints in spatially aware RBAC models. A common form of restriction is to constrain the use of roles to specific geographic locations. However, this is not the only possible use for spatially aware RBAC policies. Here, we consider the definition of policies based on the location of *other users*, rather than the one making the request. In this chapter, we define the syntax and semantics for a proximity-based policy language, based on a formal spatial model for indoor environments.

4.1 The Need for Prox-RBAC

A major limitation of current approaches to spatially aware RBAC is that they focus only on the location of the user issuing the role usage request. However in many real situations whether a user can use a role and access the resources for which access is granted to the role may depend on the presence or absence of other users. As an example, consider a government agency with data classified at multiple levels of security. One policy could prohibit access to a sensitive document if there are any civilians (*i.e.,* non-governmental employees) present. Another could require the presence of a supervisor when a document is signed. Yet another could require that the subject is alone (*e.g.,* "for your eyes only" restrictions).

In this chapter, we address the problem of specifying and enforcing a novel class of location constraints, referred to as *proximity-based* location constraints, for RBAC in both static and mobile environments. That is, we want to make decisions about granting access to roles by also taking into account the location of other users, possibly considering the proximity of those users to the requesting subject. Incorporating contextual factors in mobile environments is challenging, as these environments are inherently dynamic. As such, it is important to consider how to monitor and react as to changes in users' locations. This challenge can be described as enforcing *continuity of usage* constraints. Our approach is to adopt the policy language semantics of the UCON$_{ABC}$ family of access control models [7–9]. This family of models defines a number of semantic structures that enable the specification of contextual access control policies.

The focus of this work is to define the Prox-RBAC model and language for specifying and enforcing proximity-based location constraints. In Prox-RBAC, administrators can write policies that specify either the presence or absence of other users within a protected area. Prox-RBAC also makes a distinction between policies that require authorization only a single time prior to access and policies that specify conditions that must continue to hold for as long as the permission is used. Finally, Prox-RBAC is backward compatible; that is, Prox-RBAC can specify existing spatially aware RBAC policies, as well as traditional RBAC.

4.2 Background

As in the previous chapter, Prox-RBAC builds on the GEO-RBAC spatially aware RBAC model, as well as the UCON$_{ABC}$ family of models. Although we have already discussed the necessary background material for GEO-RBAC, we have not described the use of UCON$_{ABC}$ to define policy semantics, which is a critical piece of the Prox-RBAC language. In this section, we provide a brief summary of UCON$_{ABC}$ policies to facilitate the discussion of Prox-RBAC.

4.2.1 UCON$_{ABC}$

UCON$_{ABC}$ is a family of access control models that can be used to formalize the behavior of a system in terms of authorizations (A), obligations (B), and conditions (C), that must be satisfied either before (pre), during (on), or after (post) an access occurs. For instance, UCON$_{\text{preA}}$ can be used to formalize an access control system that requires authorizing the subject before the access is granted. With each type of model, there are multiple variations (*e.g.,* $\text{UCON}_{\text{preA}_0}$, $\text{UCON}_{\text{preA}_1}$, and $\text{UCON}_{\text{preA}_3}$). For a full description of these details, we refer the reader to the UCON_{ABC} papers [7–9].

Specifying a policy in UCON_{ABC} can be done by declaring the functions, relations, and other mathematical structures, and then defining the implication that must hold if the access is granted. As an example, consider a traditional RBAC system, where $\mathcal{S}$, $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{R}$ denote the sets of subjects, objects, actions, and roles, respectively. A permission is an ordered pair (o, a) for $o \in \mathcal{O}, a \in \mathcal{A}$. Subjects are mapped to active roles via the *ActiveRoles* function, while *PermittedRoles* maps permissions with the roles that are to be granted access. As *ActiveRoles* are user-specific, we also declare $ATT(\mathcal{S}) = \{ActiveRoles\}$, where $ATT(\mathcal{S})$ specifies the set of attributes that are associated with subjects.

To specify that a subject s is authorized to perform action a on object o, the semantics of $\text{UCON}_{\text{preA}_0}$ use an invariant $allowed(s, o, a) \Rightarrow P$, where P denotes a necessary condition for authorization. For instance, in RBAC, the necessary condition is that the requester has an active role ($\exists role \in ActiveRoles(s)$) that is granted the desired permission ($\exists role' \in PermittedRoles(o, a), role \geq role'$). Figure 4.1 illustrates how $\text{UCON}_{\text{preA}_0}$ can specify traditional RBAC policies.

$< role, act, obj > - \text{UCON}_{\text{preA}_0}$:

$$\begin{aligned}
&Perms = \{(o, a) | o \in \mathcal{O}, a \in \mathcal{A}\} \\
&ActiveRoles : \mathcal{S} \to 2^{\mathcal{R}} \\
&PermittedRoles : Perms \to 2^{\mathcal{R}} \\
&ATT(\mathcal{S}) = \{ActiveRoles\} \\
\hline
&allowed(s, o, a) \Rightarrow \exists role \in ActiveRoles(s), \exists role' \in PermittedRoles(o, a), role \geq role'
\end{aligned}$$

Figure 4.1. UCON_{ABC} semantics for traditional RBAC

To enforce that certain conditions continue to hold as the access occurs (*i.e.,* continuity of usage constraints), UCON_{ABC} defines additional primitive formalisms. First, *preCON* declares a set of conditions that are evaluated, while $getPreCON(s, o, a)$

specifies proposition built on these conditions. Then *preConChecked* can be used in the necessary proposition in the $allowed(s, o, a) \Rightarrow P$ implication. For instance, if the subject must be over the age of 18 or accompanied by an adult, we could specify this portion of the policy as shown in Figure 4.2.

$\text{UCON}_{\text{preC}_0}$:

$preCON = \{Over18(s), Accompanied(s)\}$

$getPreCON(s, o, a) = Over18(s) \vee, Accompanied(s)$

$allowed(s, o, a) \Rightarrow preConChecked(getPreCON(s, o, a))$

Figure 4.2. UCON_{ABC} semantics for mandating adult accompaniment of a minor

For on-going conditions (*i.e.,* UCON_{onC}), similar structures exist (*i.e., onCON* and $getOnCON(s, o, a)$). Permission revocation is similar to the $allowed(s, o, a)$ invariant, but with the opposite logical implication. Specifically, $stopped(s, o, a) \Leftarrow \neg onConChecked(getOnCON(s, o, a))$ declares that the access must be stopped once the on-going required condition is no longer true.

4.3 The Prox-RBAC Language

In this section we present the syntax and semantics of our proximity constraint language. After a short preliminary subsection to introduce some notation, we describe our space model, which is a key element in the definition of the proximity constraint model. We then briefly introduce our spatial role model, followed by the introduction of the three main constructs of our proximity constraint model. Such introduction is followed by the definition of the syntax and semantics of the proximity constraint model.

4.3.1 Preliminaries

The core Prox-RBAC model, which underlies our language, uses a number of primitives that are similar to existing spatially aware RBAC models. As in traditional

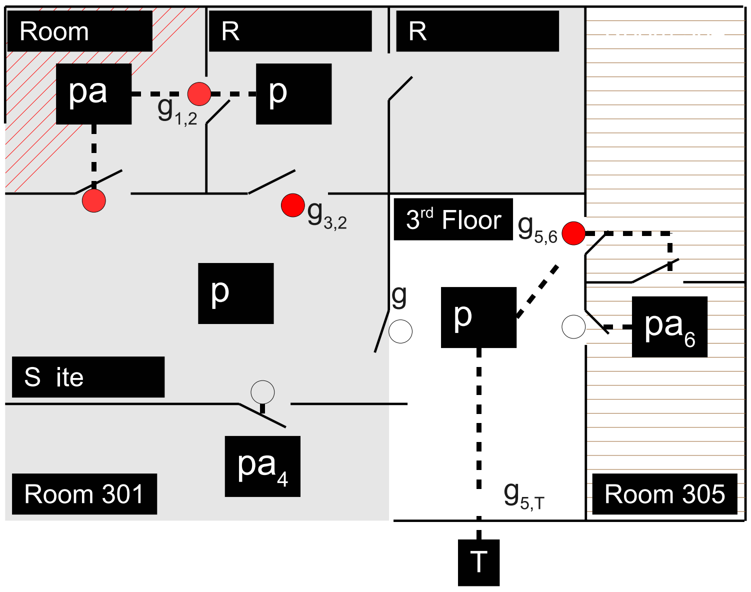


Figure 4.3. Reference space with PAs marked

RBAC, we have *Subjects* ($\mathcal{S}$) that can request permission to perform *Actions* ($\mathcal{A}$) on *Objects* ($\mathcal{O}$). Let *Roles* ($\mathcal{R}$) denote the set of roles in the system. When $s \in \mathcal{S}$ requests the privilege to perform $a \in \mathcal{A}$ on $o \in \mathcal{O}$, s must activate a role $r \in \mathcal{R}$ to which the requested permission is granted.

4.3.2 Space Model

The first challenge in defining our model is to specify how space is modeled. We adopt a spatial model that partitions generic spaces into regions based on security classifications. We consider the reference space to be a region (not necessarily bounded) that is divided into a set of **protected areas** (PA). A PA is a physically bounded region of space, accessible through a limited number of **entry points**, which consists of a physical barrier that requires authorization. Each PA can be arbitrarily large and we place no restrictions on the internal structure. For instance, a PA could consist of a single room or an entire floor that is made up of distinct but unlocked rooms. In the latter case, the subject's presence in a particular room is irrelevant to the security questions, and we model the floor in its entirety as a PA. Formally, we use $\mathcal{P}$ to denote the set of PAs in the system under consideration. We write $s \in_l pa$ to indicate that the subject $s \in \mathcal{S}$ is within the spatial extents of pa. We also write $\top \in \mathcal{P}$ to model the reference space.

Definition (Entry Point). *Let $\mathcal{P}$ be the set of protected areas and $\mathcal{G}$ be the set of entry points (or guards). Then there exists a function $g2p : \mathcal{G} \to \mathcal{P} \times \mathcal{P}$ such that:*

- $\forall g \in \mathcal{G}, \exists x, y \in \mathcal{P}$ such that $g2p(g) = (x, y)$
- $\forall g \in \mathcal{G}, g2p(g) = (x, y) \Rightarrow x \neq y$
- $\forall g \in \mathcal{G}, g2p(g) = (x, y) \Rightarrow g2p(g) = (y, x)$
- $\exists g \in \mathcal{G}, g2p(\top, x)$ for some $x \in \mathcal{P}$

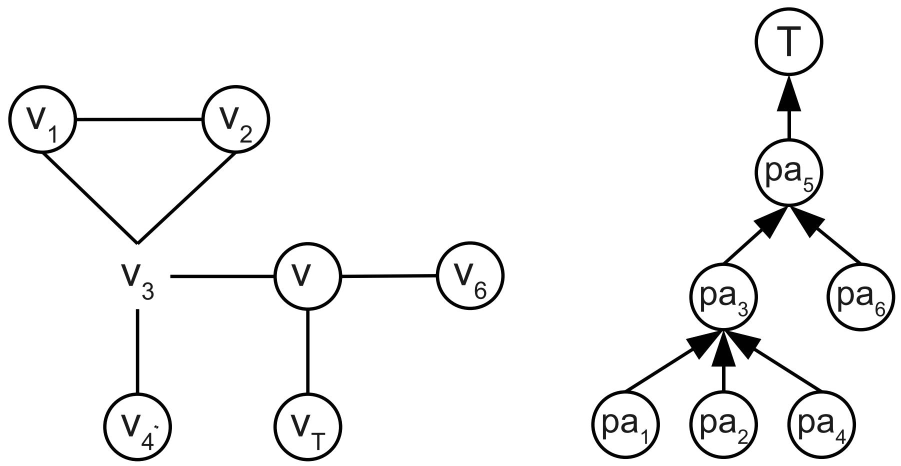
Figure 4.3 demonstrates a number of characteristics of our space. The PAs are distinguished by shading and lines filling the area. The red dots indicate the entry points and are labeled as the guard device $g_{i.j}$ controlling passage between pa_i and pa_j. Observe that rooms 303A and 303B are both part of pa_2, thus indicating that PAs do not necessarily have a one-to-one correspondence to the features of the space. In addition, suite 300A defines pa_3 and consists of the entire space shaded with gray, which includes rooms 301, 302, 303A, and 303B. Similarly, pa_5 covers the entire floor. This illustrates that PAs can be defined hierarchically. Finally, $\top$ defines the entire reference space, including all external regions not classified as PAs.

To represent this constrained space we define an indoor space model [118, 119]. Indoor space models present distinguishing features which perfectly match the requirements posed by our scenario. A major feature is that those spaces are cellular (or symbolic), *i.e.,* they consists of a finite set of named cells or symbolic coordinates (e.g., rooms 303A and 303B) [120]. As a consequence, conventional Euclidean distance and spatial network distance are inapplicable [118]. Moreover, indoor spaces may present complex topologies. Among the various topologies that one can specify, the most relevant are the connectivity between the indoor and outdoor spaces, as well as connectivity between cells (*i.e.,* PAs). We choose to define the indoor space model as the triple $(\mathcal{P}, G, H)$ where $\mathcal{P}$ is the set of names (*i.e.,* PAs in the system under consideration), G is the connectivity topology called an *accessibility graph*, and H is the hierarchy of PAs.

Definition (Accessibility Graph). *Let RS be the reference space and let $\mathcal{P}$ be the set of protected areas in RS. The accessibility graph of RS is an undirected, labeled multigraph G = (V,E). Here $V = \{v_\top, v_1, \ldots, v_n\}$ is the set of vertices with $v_\top$ corresponding to $\top$ (which represents RS), and $v_1, \ldots, v_n$ corresponding to the protected areas $pa_1, \ldots,$ $pa_n \in \mathcal{P}$, and $E = \{(v_i, v_j)\}_{[v_i, v_j \in V, i \neq j]}$ is the set of edges denoting the entry points. For $g_{i,j} \in \mathcal{G}$ such that $g2p(g_{i,j}) = (pa_i, pa_j)$, we write $e_{i,j} \in E$ for the corresponding edge in the graph. The function $p2v : \mathcal{P} \rightarrow V$ maps protected areas to vertices, where $p2v(p_i) = v_i$ for $1 \leq i \leq n$ and $p2v(\top) = v_\top$. The function $p2e : (\mathcal{P} \times \mathcal{P}) \rightarrow E$ identifies the edge for the entry point, where $p2e(pa_i, pa_j) = e_{i,j}$ if and only if there exists an edge $(p2v(pa_i), p2v(pa_j)) \in E$.*

As noted above, our model incorporates the notion of hierarchies of PAs. For instance, access to the third floor of a building may be restricted to a set of users; in addition, access to room 305 may be further restricted to an individual user. As such, when the user is in room 305, the permissions associated with the third floor should still be granted. To begin to model this hierarchy, we introduce the partial order $v_i \sqsubseteq v_j$ to indicate that pa_i is wholly contained within the region of pa_j. For instance, by mapping the PAs in Figure 4.3 to vertices, we have $v_1 \sqsubseteq v_3 \sqsubseteq v_5$. Note that, for any $v_i = v_\top$, $v_i \sqsubseteq v_\top$ and $v_\top \not\sqsubseteq v_i$. This relation also impacts our previous discussion of the user's location. Specifically, if subject $s \in_l pa_i$ (for $s \in \mathcal{S}$) and $v_i \sqsubseteq v_j$, then $s \in_l pa_j$. This partial order leads to the notion of parent.

Definition (Parent Tree). *The **parent** of a protected area $pa_i \in \mathcal{P}$, denoted Parent(pa_i), is the smallest PA enclosing pa_i. That is, Parent(pa_i) $= pa_j$ if an only if $v_i \sqsubseteq v_j$, $pa_i = pa_j$, and there does not exist pa_k such that $v_i \sqsubseteq v_k \sqsubseteq v_j$. If there*



(a) Accessibility graph (b) Parent tree

Figure 4.4. Accessibility graph and parent tree for the reference space in Figure 4.3

is no such enclosing parent of pa_i, then we write $Parent(pa_i) = \top$. If $Parent(pa_i) = pa_j$, the pa_i is a child of pa_j, and we write $pa_i \in Children(pa_j)$. The parents and children form a tree.

Figure 4.4 shows the accessibility graph and parent tree for the reference space in Figure 4.3. One challenge in applying this partial order is that spatial hierarchies are not necessarily uniform. For instance, in Figure 4.3, consider rooms 301 and 305, which are mapped to vertices v_4 and v_6, respectively. As both rooms are on the third floor and room 301 is part of suite 300A, we have $v_4 \sqsubseteq v_3 \sqsubseteq v_5$ and $v_6 \sqsubseteq v_5$. Intuitively, even though v_3 (suite 300A) and v_6 (room 305) have the same parent, they have different characteristics. To accommodate this nonuniformity, each vertex has an associated **type**. That is, we define a function $Type : V \rightarrow \tau$, where $Type(v_i) = \tau_j$ indicates that v_i has type $\tau_j \in \tau$. For instance, $floor$, $suite$, or $room$ could be types of PAs.

Finally, we must address the notion of **movement** in our model. Intuitively, the movement of a subject corresponds to the transition of the subject form one PA to another. Obviously, such a movement requires the subject's presence at an entry point. However, we must also have a way to detect that the subject did, in fact, pass through the entry point, rather than doubling back into the current space. Consequently, we define the notion of movement as follows.

Definition (Movement). *Let RS be the reference space and AG be the accessibility graph for RS. The movement $M(s, pa_i, pa_j)$ in AG is the traversal of the edge $(e_i, e_j) \in E$ such that subject s is authenticated and authorized to enter pa_j, and the subject's entry into the new PA is confirmed.*

Key to our definition of movement is the authentication and confirmation of passage. That is, enforcement of Prox-RBAC requires a physical barrier separating PAs, as well as deployment of a technology that ensures only authorized personnel can pass. Enforcement does not require close monitoring of users' locations within PAs, but maintaining an accurate mapping of users to PAs is critical for security. As we will describe in Section 9.5, detecting movement is a challenging issue in deployment of spatially aware RBAC systems.

4.3.3 Spatial Roles

As in GEO-RBAC, we augment traditional roles with geographic information. Specifically, a **spatial role** is the tuple $< r, pa >$, where $r \in \mathcal{R}$ and $pa \in \mathcal{P} \cup \{\top\}$. The intuition is that a spatial role $< r, pa >$ is **enabled** for the subject s if s is authorized to activate the traditional role r, and $s \in_l pa$. The spatial role is then **activated** if the user wishes to exercise the privileges associated with the role.

Permitting continuous access in Prox-RBAC entails addressing the challenge of reacting to changes as the user moves. Specifically, it is not obvious how to handle role activation when the user moves to a new PA.While one approach would be to require re-activation of roles as the user moves, we adopt a different approach to streamline the user experience. Consider the movement $M(s, pa_i, pa_j)$. During the transition, $s \notin_l pa_i$ and $s \notin_l pa_j$. However, $s \in_l pa_k$, where pa_k is the **least common parent**. That is, $pa_i \sqsubseteq pa_k$, $pa_j \sqsubseteq pa_k$, and either $pa_i \not\sqsubseteq pa_l$ or $pa_j \not\sqsubseteq pa_l$ for any $pa_l \in Children(pa_k)$. During the transition, the active role $< r, pa_i >$ will be updated to $< r, pa_k >$. As a result, the permissions that s currently holds may change. Once s has confirmed pa_j as his new location, the roles will again be updated to $< r, pa_j >$.

4.3.4 Proximity Constraints

A **proximity constraint** is a security requirement that is satisfied by the location of other users. Proximity constraints are built from three primitive constructs: relative constraint clauses, continuity of usage constraints, and timeouts. Relative constraint clauses define the static presence or absence conditions that must be met. However, mobile environments are inherently dynamic. As such, the latter two constructs are necessary to ensure the relative constraint clause is enforced properly as the environment changes.

A **relative constraint clause** specifies the proximity requirement of other users in the spatial environment. These clauses can be described as either **presence constraints** or **absence constraints**. To formulate these conditions, we adopt an intuitive syntax that can be illustrated as follows:

$$at\_most\ 0\ civilian\ in\ Room305$$

The basic structure consists of an optional cardinality qualifier (*e.g., at_most* or *at_least*), a nonnegative integer specifying the number of subjects, a role (*e.g., civilian*), and a spatial relationship (*e.g., in Room*305). The spatial relationship consists of two parts: a topological relation and a logical location descriptor that identifies a PA. Let $\mathcal{R}_T$ denote the set of topological relations. In our initial approach, we will only consider a small set of relations, name $\mathcal{R}_T = \{in, out, adj\}$. The location descriptor can be absolute, as was the case here, or it can take the form of *this.space*. In this latter structure, the *space* specifies a level in the hierarchy of PAs, while *this* dictates that the location of the subject fulfilling the role in the clause must match that of the requester. For instance, let v_i denote $Room305$, v_j denote $Room300$, and v_k denote $Floor3$. Assume the requester is in $Room305$ and his supervisor is in $Room300$. Since $v_i \sqsubseteq v_k$ and $v_j \sqsubseteq v_k$, the following relative constraint clause would be satisfied:

$$at\_least\ 1\ supervisor\ in\ this.floor$$

Some operations may require a significant duration. For instance, reading a sensitive document may take several minutes or hours. Furthermore, it may be necessary to ensure the relative constraint holds for the entire duration of the permitted session. To declare whether the constraint must be checked only at the beginning of the session or must hold for the duration, we introduce into our language **continuity of usage qualifiers**, called *when* and *while*, respectively. Their use is illustrated as follows:

$$while\ (\ at\_most\ 0\ civilian\ in\ Room305\ )$$

A **when constraint** is evaluated at the access request time; if the constraint is satisfied, the permission is granted. A **while constraint** is repeatedly checked and the permission is granted until the constraint is violated. The frequency of the check is a system-wide parameter that is dependent on the deployment scenario. That is, specifying this parameter requires considering issues such as network latency, size of the spatial environment, number and mobility of users.

In many cases, the desired security guarantees may require satisfying multiple relative constraints. To allow for such cases, we permit the use of two basic logical connectives: $\vee$ (logical or) and $\wedge$ (logical and). These logical connectives can be used to join relative constraint clauses or continuity of usage constraints. Parentheses may be used to specify precedence; otherwise, the clauses are enforced left-to-right. As an example, assume that c_1 is a *while* constraint dictating that no civilians are present. In addition, either a supervisor or accountant must initially be present (c_2 or c_3). The following constraints are equivalent in expressing this requirement:

$$while\ (\ c_1\ )\ \wedge\ (\ when\ (\ c_2\ )\ \vee\ when\ (\ c_3\ )\ )$$
$$while\ (\ c_1\ )\ \wedge\ when\ (\ c_2\ \vee\ c_3\ )$$

One critical issue in enforcing continuity of usage constraints is how to react once a *while* constraint no longer holds. In one scenario, the permission could be suspended until the condition is once again satisfied. In others, it may be acceptable to allow some leeway, wherein the permission is still granted for a short duration of time, even though the condition is technically being violated. For instance, consider a proximity constraint that specifies a supervisor must be present to read an accounting record. Due to a shift change, one supervisor leaves the room before the next arrives. However, the break is short enough that it is acceptable to allow the subject to retain the permission during their absences.

In some cases, it is acceptable for proximity constraints to be violated for a brief duration. For instance, if the policy specifies the presence of a supervisor, it would be undesirable for the employee to lose permissions while the supervisor leaves for a short break. Consequently, every proximity constraint that includes at least one *while* clause must end with a **timeout constraint**, which takes the following form:

$$\textit{while} \ (\ \textit{clause} \ ) \ \textit{timeout} \ t$$

Here, $t \in \mathbb{N}_0$ specifies the maximum amount of time for which the permission is granted once the *while* constraint fails. While the simplest approach is to use a single time unit for all timeout constraints, a straightforward augmentation of our language could allow t to specify the units, as well. If $t = 0$, then the permission is immediately revoked. If the condition is once again satisfied before the time limit has been reached, the permission is automatically extended as if the condition held for the entire duration.

4.3.5 Prox-RBAC Syntax

To formalize this syntax,[1] let $\mathcal{C}$ denote the set of basic relative constraint clauses with no Boolean connectives. Formally, we can write $c =< q, n, r, r_t, p >$, where q is

[1]For the sake of simplicity, we omit from our grammars any parentheses that can be used to indicate Boolean formulas. We feel that including them in the specification needlessly complicates the discussion and distracts the reader from the most relevant topics.

a cardinality qualifier, $n \in \mathbb{N}_0$, $r \in \mathcal{R}$, $r_t \in \mathcal{R}_T$, and $p \in \mathcal{P}$. $\mathcal{C}^*$, then, denotes the set of clauses that can be constructed from a Boolean formula of basic clauses. That is, for $c \in \mathcal{C}^*$, $c =< c_0, b_1, c_1, \ldots, b_n, c_n >$ for $b_i \in \{\vee, \wedge\}$ for $1 \leq i \leq n$ and $c_j \in \mathcal{C}$ for $0 \leq j \leq n$. This produces the following grammar for constraint clauses:

$$\begin{array}{rcl} C & ::= & C \vee C \\ & | & C \wedge C \\ & | & Q \;\; n \;\; role \;\; topo \;\; pa \\ Q & ::= & at\_most \\ & | & at\_least \\ & | & \epsilon \end{array}$$

Now, let $\mathcal{W}$ denote the set of continuity of usage constraints. That is, *while* (c_i) $\in \mathcal{W}$ and *when* (c_j) $\in \mathcal{W}$ if $c_i, c_j \in \mathcal{C}^*$. Given that timeouts can only apply to *while* constraints, we create a distinguished set $\mathcal{W}_{while} \subseteq \mathcal{W}$ that consists exclusively of the constraints *while* (c_i), where $c_i \in \mathcal{C}^*$. As before, let $\mathcal{W}^*$ denote the set of Boolean conjunctions and disjunctions that can be formed from any combination of continuity of usage constraints.[2] As above, for any $w \in \mathcal{W}^*$, w can be written as the tuple $w =< w_0, b_1, w_1, \ldots, b_n, w_n >$, where $b_i \in \{\vee, \wedge\}$ for $1 \leq i \leq n$ and $w_j \in \mathcal{W}$ for $0 \leq j \leq n$. This leads to the following grammar rules:

$$\begin{array}{rcl} W & ::= & W \wedge W \\ & | & W \vee W \\ & | & while\ C \\ & | & when\ C \end{array}$$

Finally, let $\mathcal{T}$ denote the set of timeouts (*i.e., timeout* t, where t may specify the time unit if permitted by the system designers). In addition, $\epsilon \in \mathcal{T}$. Then a proximity

[2]Observe that negations are not necessary in our language. First, negations would only be applicable in joining relative constraint clauses (*i.e.,* statements such as *not while* (c) would be awkward). Second, the *at_most* and *at_least* qualifiers are clear opposites (*e.g., at_most* 0 is the negation of *at_least* 1). Thus, our language can express negations without introducing an explicit Boolean operator.

constraint can be written as $< w, t > \in \mathcal{W}^* \times T$. If $w =< w_0, b_1, w_1, \ldots, b_n, w_n >$, then $t = \epsilon$ if $w_i \notin \mathcal{W}_{while}$ for $0 \leq i \leq n$ (*i.e.,* w consists exclusively of *when* constraints). We write $\Phi = (\mathcal{W}^* \times T) \cup \{\perp\}$ to denote the set of all possible proximity constraints. That is, $\perp$ denotes the absence of a proximity constraint, which allows for traditional spatially aware policies.

4.3.6 Prox-RBAC Policies and Semantics

A Prox-RBAC **policy** is a tuple of the form $< sr, a, o, \varphi >$, where $sr \in \mathcal{R} \times \{\mathcal{P} \cup \{\top\}\}$, $a \in \mathcal{A}$, $o \in \mathcal{O}$, and $\varphi \in \Phi$. In this section, we present the formal semantics for Prox-RBAC in terms of the UCON$_{ABC}$ family of core models. Prox-RBAC employs $UCON_{AC}$ semantics, as we require authorizations (A) and conditions (C), but not obligations (B). In all of the semantic specifications below, $\geq$ can denote either the dominance relation on the partially ordered set of roles $\mathcal{R}$ or the traditional inequality on integers. The notation 2^S refers to the power set of the set S.

We start with the simplest case, in which $\varphi = \perp$. That is, there is no proximity constraint enforced, and the policy indicates a spatially aware RBAC role as defined in existing works. We can write these semantics formally as a UCON$_{\text{preA}_0}$ policy, as shown in Figure 4.5.

$< role, act, obj, \perp > -$ UCON$_{\text{preA}_0}$:

$$
\begin{aligned}
&role =< r, pa >, r \in \mathcal{R}, pa \in \mathcal{P} \\
&Perms = \{(o, a) | o \in \mathcal{O}, a \in \mathcal{A}\} \\
&ActiveRoles : \mathcal{S} \rightarrow 2^{\mathcal{R}} \\
&EnabledRoles : \mathcal{P} \rightarrow 2^{\mathcal{R}} \\
&PermittedRoles : Perms \rightarrow 2^{\mathcal{R}} \\
&ATT(\mathcal{S}) = \{ActiveRoles\}
\end{aligned}
$$

$$
\begin{aligned}
&allowed(s, o, a) \Rightarrow \exists\ r \in EnabledRoles(pa), s \in_l pa \wedge r \in ActiveRoles(s)\ \wedge \\
&\quad \exists\ r' \in PermittedRoles(o, a), r \geq r'
\end{aligned}
$$

Figure 4.5. Prox-RBAC semantics for policies with no proximity constraint

These semantics state that, if s is allowed to perform a on o, there must be a traditional RBAC role r that is enabled by entering pa, s is physically present there, and s has activated the role. In addition, r must dominate r', which is a traditional RBAC role that is permitted to perform a on o. Obviously, it may be the case that $r = r'$. However, when hierarchical roles are created, $r \geq r'$ implies that activating r inherits all of the permissions associated with r'. As this is $\text{UCON}_{\text{preA}_0}$, this policy is checked only once prior to granting access. Finally, note that the same semantics can be applied for traditional RBAC policies, as well. In that case, $pa = \top$, which indicates that role activation can occur anywhere. Thus, Prox-RBAC semantics are flexible enough to accommodate more traditional policies.

To define the semantics for *when* and *while* constraints, we must define a number of helper functions, which are listed in Table 4.1. $TopoSat$ defines the conditions under which topological relations are satisfied. *in* is satisfied if the subject is inside the PA, while *out* is satisfied if the subject is not in the PA. *adj* is satisfied if the subject is in an adjacent PA. This function is used in the context of the Sat function, which finds a set of subjects that have r as an active role and satisfy the topological relation. $AtMostSat$ and $AtLeastSat$ are intuitive variations.

Building on these functions, we can define what it means to satisfy a relative constraint clause $c \in \mathcal{C}^*$. Recall that c can either be a simple clause ($c \in \mathcal{C}$)), or it can be a complex clause that is created from disjuctions and/or conjunctions of simple clauses. That is, $c =< c_0, b_1, c_1, \ldots, b_n, c_n >$, where c_i is a simple clause and b_i is a Boolean connective. Let $c_i.q$ denote the quantifier (*i.e.*, *at_least*, *at_most*, or ϵ) for the simple clause c_i. Similarly, let $c_i.n, c_i.r, c_i.r_t$, and $c_i.p$ denote the remaining portions of the clause. If we let $\mathcal{B}$ denote the set of Boolean clauses of the form $< p_0$, b_1, p_1, $\ldots, b_n, p_n >$, where $p_i \in \{true, false\}$ for $0 \leq i \leq n$ and $b_j \in \{\wedge, \vee\}$ for $1 \leq j \leq n$, we can define the $ConToBool$ function to convert a relative constraint clause to a Boolean clause.

The last component we need for relative constraint clause satisfcation is $SubThis$, which is a function for handling the *this* keyword. Specifically, $SubThis$ will traverse

Table 4.1
Helper functions for evaluating Prox-RBAC semantics

$\underline{TopoSat : \mathcal{S} \times \mathcal{R}_T \times \mathcal{P} \rightarrow \{true, false\}}$

$TopoSat(s, r_t, p) = true$ if and only if one of these hold:

$r_t = in$ and $\exists p' \in \mathcal{P}\ (s \in_l p' \wedge p2v(p') \sqsubseteq p2v(p))$

$r_t = out$ and $\exists p' \in \mathcal{P}\ (s \in_l p' \wedge p2v(p') \not\sqsubseteq p2v(p))$

$r_t = adj$ and $\exists p' \in \mathcal{P}\ (s \in_l p' \wedge (p2v(p'), p2v(p)) \in E)$

$\underline{Sat : (\mathbb{N}_0) \times \mathcal{R} \times \mathcal{R}_T \times \mathcal{P} \rightarrow \{true, false\}}$

$Sat(n, r, r_t, p) = true$ if and only if $\exists S' \subseteq \mathcal{S}, |S'| = n$

$\forall s \in S'\ \exists role \in ActiveRoles(s)$ such that

$(r = role \wedge TopoSat(s, r_t, p))$

$\underline{AtMostSat : (\mathbb{N}_0) \times \mathcal{R} \times \mathcal{R}_T \times \mathcal{P} \rightarrow \{true, false\}}$

$AtMostSat(n, r, r_t, p) = true$ if and only if $\forall S' \subseteq \mathcal{S}, |S'| \leq n$

whenever $\forall s \in S'\ \exists role \in ActiveRoles(s)$ such that

$(r = role \wedge TopoSat(s, r_t, p))$

$\underline{AtLeastSat : (\mathbb{N}_0) \times \mathcal{R} \times \mathcal{R}_T \times \mathcal{P} \rightarrow \{true, false\}}$

$AtLeastSat(n, r, r_t, p) = true$ if and only if $\exists S' \subseteq \mathcal{S}, |S'| \geq n$

$\forall s \in S'\ \exists role \in ActiveRoles(s)$ such that

$(r = role \wedge TopoSat(s, r_t, p))$

$\underline{SubThis : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{C}}$

$SubThis(c, pa) = < c'_0, b_1, c'_1, \ldots, b_n, c'_n >$, where

$\forall i, 0 \leq i \leq n,$

if $c_i.p = this.\tau_j \wedge CastType(pa, \tau_j) = pa'$, then $c'_i.p = pa'$

else $c'_i.p = c_i.p$

$\underline{ConToBool : \mathcal{C} \rightarrow \mathcal{B}}$

$ConToBool(c) = < p_0, b_1, p_1, \ldots, b_n, p_n >$, where

$\forall i, 0 \leq i \leq n, p_i = true$ if and only

$[\ (c_i.q = at\_least \wedge AtLeastSat(c_i.n, c_i.r, c_i.r_t, c_i.p)) \vee$

$(c_i.q = at\_most \wedge AtMostSat(c_i.n, c_i.r, c_i.r_t, c_i.p)) \vee$

$(c_i.q = \epsilon \wedge Sat(c_i.n, c_i.r, c_i.r_t, c_i.p))\ ]$

$\underline{ConSat : \mathcal{C} \times \mathcal{P} \rightarrow \{true, false\}}$

$ConSat(c, pa) = BoolSat(ConToBool(SubThis(c, pa)))$

$\underline{WToBool : \mathcal{W}^* \times \mathcal{P} \rightarrow \mathcal{B}}$

$WToBool(w, pa) = < p_0, b_1, p_1, \ldots, b_n, p_n >$, where

$\forall i, 0 \leq i \leq n, p_i = ConSat(w_i.c, pa)$

$\underline{WToWhileOnly : \mathcal{W}^* \rightarrow \mathcal{W}^*}$

$WToWhileOnly(< w_0, b_1, w_1, \ldots, b_n, w_n >= \epsilon$ if

$w_i \notin \mathcal{W}_{while}$ for $0 \leq i \leq n$

$WToWhileOnly(< w_0, b_1, w_1, \ldots, b_n, w_n >) =$

$< w'_0, b_1, w'_1, \ldots, b_m, w'_m >$, where

$(m \leq n \wedge$ for $0 \leq i, k \leq n, 0 \leq j, j \leq m$

$(\ (w_i \in \mathcal{W}_{while} \Rightarrow \exists w'_j = w_i) \wedge$

$(w'_j \in \mathcal{W}_{while}) \wedge$

$((w_i = w'_j \wedge w_k = w'_l \wedge i \leq k) \Rightarrow j \leq l) \wedge$

$(w_i \in \mathcal{W}_{while} \Rightarrow (\exists b'_j = b_i \Leftarrow \exists w'_l, l < j)) \wedge$

$((b_i = b'_j \wedge b_k = b'_l \wedge i \leq k) \Rightarrow j \leq l)\ )$

the clause c, replacing $this.\tau_j$ with pa, assuming $Type(pa) = \tau_j$. If $Type(pa) < \tau_j$, then the parent tree is traversed until the appropriate type is found. We use $CastType(pa, \tau_j)$ to denote the (unspecified) function that handles this traversal. Note that pa starts at the finest granularity, so $Type(pa) = \tau_j$. Now, let $BoolSat : \mathcal{B} \to \{true, false\}$ denote a function that evaluates a Boolean clause. We can assemble all of these components to define relative constraint clause satisfaction with the $ConSat$ function, where pa is the subject's PA.

We are now ready to specify semantics for proximity constraints consisting exclusively of *when* constraints. Let $c =< w, \epsilon >$, where $w =< w_0, b_1, w_1, \ldots, b_n, w_n >$, $w_i \in \mathcal{W}$, and $w_i \notin \mathcal{W}_{while}$ for $0 \leq i \leq n$. Let $w_i.c$ denote the relative constraint clause for the *when* clause w_i. Similar to *ConToBool*, we introduce *WToBool* to convert a set of *when* or *while* constraints to a Boolean formula. Using this formula, we can specify the semantics for *when* constraints as a $\text{UCON}_{\text{preC}_0}$ policy, meaning that the condition and authorization are checked only prior to authorization, as shown in Figure 4.6.

$< role, act, obj, when > - \text{UCON}_{\text{preC}_0}$:

$$\begin{aligned}
&WToBool : \mathcal{W}^* \times \mathcal{P} \to \mathcal{B} \\
&BoolSat : \mathcal{B} \to \{true, false\} \\
&Loc : \mathcal{S} \to \mathcal{P} \\
&role =< r, pa >, r \in \mathcal{R}, pa \in \mathcal{P} \\
&Perms = \{(o, a) | o \in \mathcal{O}, a \in \mathcal{A}\} \\
&ActiveRoles : \mathcal{S} \to 2^{\mathcal{R}} \\
&EnabledRoles : \mathcal{P} \to 2^{\mathcal{R}} \\
&PermittedRoles : Perms \to 2^{\mathcal{R}} \\
&ATT(\mathcal{S}) = \{ActiveRoles\} \\
&preCON = \{BoolSat(WToBool(when, Loc(s)))\} \\
&getPreCON(s, o, a) = BoolSat(WToBool(when, Loc(s))) \\
\hline
&allowed(s, o, a) \Rightarrow preConChecked(getPreCON(s, o, a)) \wedge \\
&\quad (\exists\, r \in EnabledRoles(pa), s \in_l pa \wedge r \in ActiveRoles(s) \wedge \\
&\quad\ \exists\, r' \in PermittedRoles(o, a), r \geq r')
\end{aligned}$$

Figure 4.6. Prox-RBAC semantics for policies with *when* constraints

While constraints with a timeout of 0 (*i.e.,* immediate revocation) are very similar to *when* constraints, with the exception that a subset of the conditions are repeatedly checked as the permission is exercised. To model this behavior, we introduce the *WToWhileOnly* function that converts a sequence of *when* and *while* constraints to the corresponding list that only contains *while* constraints. If there were no *while* constraints, *WToWhileOnly* would return ϵ, indicating there are no on-going conditions to enforce. Using this function, Figure 4.7 specifies the semantics of a *while* constraint with immediate revocation as $\mathrm{UCON}_{\mathrm{preC_0onC_0}}$.

$< role, act, obj, while > -\ \mathrm{UCON}_{\mathrm{preC_0onC_0}}$:

$$
\begin{aligned}
&WToWhileOnly : \mathcal{W}^* \to \mathcal{W}^* \\
&WToBool : \mathcal{W}^* \times \mathcal{P} \to \mathcal{B} \\
&BoolSat : \mathcal{B} \to \{true, false\} \\
&Location : \mathcal{S} \to \mathcal{P} \\
&role =< r, pa >, r \in \mathcal{R}, pa \in \mathcal{P} \\
&Perms = \{(o,a) | o \in \mathcal{O}, a \in \mathcal{A}\} \\
&ActiveRoles : \mathcal{S} \to 2^{\mathcal{R}} \\
&EnabledRoles : \mathcal{P} \to 2^{\mathcal{R}} \\
&PermittedRoles : Perms \to 2^{\mathcal{R}} \\
&ATT(\mathcal{S}) = \{ActiveRoles\} \\
&preCON = \{BoolSat(WToBool(while, Loc(s)))\} \\
&getPreCON(s,o,a) = BoolSat(WToBool(while, Loc(s))) \\
&onCON = \{BoolSat(WToBool(WToWhileOnly(while), Loc(s)))\} \\
&getOnCON(s,o,a) = BoolSat(WToBool(WToWhileOnly(while), Loc(s)))
\end{aligned}
$$

$$
\begin{aligned}
&allowed(s,o,a) \Rightarrow \\
&\quad preConChecked(getPreCON(s,o,a)) \wedge \\
&\quad (\exists\ r \in EnabledRoles(pa), s \in_l pa \wedge r \in ActiveRoles(s)\ \wedge \\
&\qquad \exists\ r' \in PermittedRoles(o,a), r \geq r') \\
&stopped(s,o,a) \Leftarrow \neg onConChecked(getOnCON(s,o,a))
\end{aligned}
$$

Figure 4.7. Prox-RBAC semantics for *when* constraints with immediate revocation

The final type of constraint to consider is a *while* constraint with a timeout $t > 0$. Unfortunately, UCON does not define the necessary semantic structure. Specifically, for authorizations (UCON_{onA}) and obligations (UCON_{onB}), the family defines procedures for modifying the attributes of the subject during access. However, UCON_{onC} does not have such a procedure. Thus, UCON_{onC} offers no way to update the sub-

$< role, act, obj, while, time > -$ UCON$_{\mathrm{preC_0onC_0}}$:

$CurentTime \in \mathbb{R}$ is the current system time
$PermExp : \mathcal{S} \rightarrow \mathcal{E}$
$UpdateExp : \mathcal{E} \times \mathcal{O} \times \mathcal{A} \times \mathbb{Z}^+ \times \mathbb{Z}^+ \rightarrow \mathcal{E}$
$FindExp : \mathcal{E} \times \mathcal{O} \times \mathcal{A} \rightarrow \mathbb{Z}^+$
$WToWhileOnly : \mathcal{W}^* \rightarrow \mathcal{W}^*$
$WToBool : \mathcal{W}^* \times \mathcal{P} \rightarrow \mathcal{B}$
$BoolSat : \mathcal{B} \rightarrow \{true, false\}$
$Location : \mathcal{S} \rightarrow \mathcal{P}$
$Perms = \{(o, a) | o \in \mathcal{O}, a \in \mathcal{A}\}$
$ActiveRoles : \mathcal{S} \rightarrow 2^{\mathcal{R}}$
$EnabledRoles : \mathcal{P} \rightarrow 2^{\mathcal{R}}$
$PermittedRoles : Perms \rightarrow 2^{\mathcal{R}}$
$ATT(\mathcal{S}) = \{ActiveRoles, PermExp\}$
$preCON = \{BoolSat(WToBool(while, Loc(s)))\}$
$getPreCON(s, o, a) = BoolSat(WToBool(while, Loc(s)))$
$onCON = \{BoolSat(WToBool(WToWhileOnly(while), Loc(s)))\}$
$getOnCON(s, o, a) =$
$\quad (BoolSat(WToBool(WToWhileOnly(while), Loc(s))) \vee$
$\quad\quad CurrentTime \leq FindExp(PermExp(s), o, a))$

$allowed(s, o, a) \Rightarrow$
$\quad preConChecked(getPreCON(s, o, a)) \wedge$
$\quad (\exists\, r \in EnabledRoles(pa), s \in_l pa \wedge r \in ActiveRoles(s) \wedge$
$\quad\quad \exists\, r' \in PermittedRoles(o, a), r \geq r')$
$onUpdate(PermExp(s)) : PermExp(s) =$
$\quad UpdateExp(PermExp(s), o, a, t, time)$ if
$\quad\quad BoolSat(WToBool(WToWhileOnly(while), Loc(s)))$
$stopped(s, o, a) \Leftarrow \neg onConChecked(getOnCON(s, o, a))$

Figure 4.8. Prox-RBAC semantics for *while* constraints with later expiration

ject's attributes to capture the last time the condition was true. As this procedure does not exist, we cannot specify the semantics for *while* with *timeout* using the UCON$_{ABC}$ model, strictly speaking. However, it is straightforward to adapt the *onUpdate*$(ATT(s))$ procedure from other portions of UCON$_{ABC}$ for these purposes.

To start, we introduce $\mathcal{E}$ to denote the data structures containing expiration times for permissions. That is, $PermExp(s)$ will return $e \in \mathcal{E}$. The simplest form of e would be a 2-dimensional array $\mathcal{O} \times \mathcal{A}$, where (o, a) would store the expiration time for exercising $a \in \mathcal{A}$ on $o \in \mathcal{O}$. As such, we use the notation $e[o, a]$ for this value, though we formalize this behavior with the $FindExp$ function. The $UpdateExp$ function takes such a data structure, and updates only the $e[o, a]$ entry to be $t_c + t_e$, the sum of the current system time and the expiration time. All other entries remain unchanged.

Using these functions, we extend the $\mathrm{UCON}_{\mathrm{preC_0onC_0}}$ model to express *while* constraints with a *timeout* using the following semantics. Our extension is to introduce the *onUpdate*$(ATT(s))$ procedure to update the subject's $PermExp$ attribute as the access occurs. During access, if the condition holds (*i.e.,* the *while* clause is satisfied), then the expiration is updated accordingly. The permission is revoked only if the condition fails *and the current time is greater than the expiration time.* The semantics are shown in Figure 4.8.

4.4 Enforcement Architecture

Our architectural design couples a centralized authorization server with distributed, asynchronous clients. These clients can be either stationary (*e.g.,* workstations) or mobile devices (*e.g.,* laptops). In either case, accessing sensitive data requires a physical connection to a fixed-location device, such as an near-field communication (NFC) or magnetic stripe reader. In addition, each entry point requires a fixed-location reader in each PA. The following section defines the assumed properties of each such *principal.*

4.4.1 Principals

The principals for enforcing Prox-RBAC are as follows.

- *Authorization Server (AS)* – the centralized server that acts as the policy decision point (PDP). The *AS* maintains a mapping of all user's locations, relative to a PA. The *AS* also maintains all access control policies.

- *Guard* – a fixed-location reader at an entry point. The guard is responsible for controlling a physical barrier, such as a locked door. We denote the guard that is physically located in pa_i and controls access to pa_j as $g_{i,j}$. Each guard is provided a certificate $Cert(g_{i,j})$, which is signed by the AS and contains a public key $pk(g_{i,j})$ and coordinates $cdt(g_{i,j})$. Every guard is equipped with three types of communication technologies. First, there is a network connection (either wired or over encrypted wireless) that allows for communication with the AS. Second, guards $g_{i,j}$ and $g_{j,i}$ have a physical link that allows direct communication for synchronizing control over the entry between the two PAs. Third, each guard employs a technology that requires close physical proximity for communication. For instance, guards could be a card reader or could use NFC.

- *Client* – a trusted computing device for accessing sensitive data. The client acts as the policy enforcement point, revoking privileges according to the decision of the AS. As clients may be mobile, they are identified solely by an identifier and denoted c_i. Each client has a network connection to communicate with the AS. Also, each client must be equipped with a trusted computing component (TCC), such as a TPM. That is, if an application is granted access to sensitive data, the data will be encrypted with a symmetric key that is bound to that application. To transfer the symmetric key between the client and the AS, the TCC can generate a public-private key pair, denoted $pk(c_i)$ and $sk(c_i)$ respectively. To guarantee the provenance of these keys, the TCC is equipped with a persistent key, such as the Endorsement Key in a TPM, that is used for signing certificates $Cert(c_i)$ that distribute $pk(c_i)$. We assume that the AS is able to authenticate all such certificates. Additionally, we assume unauthorized software is prevented from accessing sensitive data, and remote attestation techniques are used to ensure that the software on the client matches a pre-approved configuration.

Table 4.2
Cryptographic primitives

$\mathsf{Prove}(\cdot)$ – an interactive zero-knowledge proof of knowledge protocol.
$\mathsf{Enc}_k(\cdot)$ – a (t, ϵ)-secure symmetric key cipher that is resilient against probabilistic polynomial-time (PPT) adversaries.
$\mathsf{Enc}_{pk(p)}(\cdot)$ – encryption using the public key of p that provides indistinguishability against chosen-ciphertext attacks (IND-CCA).
$\mathsf{H}(\cdot)$ – a collision-resistent hash function.
$\mathsf{Sign}_{sk(p)}(\cdot)$ – digitally sign a message using the private key of p, such that the signature scheme provides IND-CCA security.
$\mathsf{Commit}(c, \cdot)$ – compute a commitment c, such that the commitment scheme provides unconditional hiding and computational binding.
$\mathsf{Open}(c, \cdot)$ – open the commitment c by revealing the data necessary for an honest verifier.
$\mathsf{Auth}(\cdot)$ – perform an interactive authentication protocol to establish the subject identity.

- *Location Device* – a fixed-location reader distributed in a PA. These devices are used to authenticate the location of the user at the time of an access request. Since these devices have fixed locations, each is denoted $ld_{i,j}$ to indicate the j^{th} location device in pa_i. As with guards, location devices employ a proximity-based communication technology and possess a certificate $Cert(ld_{i,j})$ that is signed by the *AS*. These certificates contain coordinates $cdt(ld_{i,j})$, as well as a public commitment $cmt(sv(ld_{i,j}))$ where $sv(ld_{i,j})$ is a secret value stored on $ld_{i,j}$. In order to bind the client to a location, it must have a physical connection to a location device.

- *User* – the human user requesting access. Each user (subject) s will have an identifier id_s, a password pwd_s and a proximity-connection device that is used for communicating with the guards and location devices. This device will store a certificate, signed by the *AS*, that contains data required to prove knowledge of a secret value $sv(s)$, which is stored on the device.

4.4.2 Protocols

Proper enforcement of Prox-RBAC requires a high-level of assurance that the system has an accurate map of users' locations. To accomplish this goal, we propose the following protocols that authenticate all requests and movements. These protocols utilize a number of cryptographic primitives. Our convention is to use $pk(p)$ and $sk(p)$ to refer to the public and secret keys, respectively, of principal p when an asymmetric cipher is required, while k denotes a private symmetric key. Using these conventions, our primitives are listed in Table 4.2.

Our first protocol, $\mathsf{Read}(s, o, r, c_i, ld_{j,k})$, is shown in Table 4.3. It is initiated by a subject $s \in \mathcal{S}$ to request read permission on object $o \in \mathcal{O}$, while using role r on client c_i at location $ld_{j,k}$. The protocol starts by s using his proximity device to prove knowledge of $sv(s)$ to $ld_{j,k}$ and entering id_s, pwd_s, and r into the client via a trusted path. The client presents a signed version of id_s to $ld_{j,k}$ via the physical connection, and $ld_{j,k}$ responds with a commitment c, binding id_s to c_i at timestamp T with a nonce n. Note that only $ld_{j,k}$ is able to open this commitment. c_i signs the commitment, sending the result to the AS. The AS and c_i then enter an authentication protocol that confirms the identity of id_s and his authorization to enter r.

Assuming the authentication of s is successful, the AS returns a signed version of the commitment c, encrypted in a manner that is only readable by c_i. c_i decrypts the signed commitment, and forwards the result to $ld_{j,k}$, who confirms the signature of the AS (thus indicating that the AS received the commitment intact). $ld_{j,k}$ opens the commitment and encrypts the result with the public key of the AS. c_i forwards the encrypted packet and the name of the object o requested, and the AS returns the encrypted object with a key bound to c_i.

Informally, we can identify a number of strengths of this protocol. First, the commitment scheme, the encrypted version of the opening, and the required physical connection between c_i and $ld_{j,k}$ ensure that the AS has a strong assurance that s is, in fact, at the location of $ld_{j,k}$ at time T. Next, the use of a TCC for c_i binds the

Table 4.3
Protocol for requesting read access

$\mathsf{Read}(s, o, r, c_i, ld_{j,k})$ – Subject s activates role r and requests *read* privilege on object o, using client c_i at location device $ld_{j,k}$.		
(1)	$s \to ld_{j,k}$	$\mathsf{Prove}(sv(s))$
(2)	$s \to c_i$	id_s, pwd_s, r
(3)	$c_i \to ld_{j,k}$	$\mathsf{Sign}_{sk(c_i)}(id_s), Cert(c_i)$
(4)	$ld_{j,k} \to c_i$	$\mathsf{Commit}(c, \mathsf{H}(n, id_s, T, pk(c_i), sv(ld_{j,k})))$
(5)	$c_i \to AS$	$\mathsf{Sign}_{sk(c_i)}(c), Cert(c_i)$
(6)	$c_i \leftrightarrow AS$	$\mathsf{Auth}(id_s, pwd_s, r)$
(7)	$AS \to c_i$	$\mathsf{Enc}_{pk(c_i)}(k), \mathsf{Enc}_k(\mathsf{Sign}_{sk(AS)}(c, pk(c_i))$
(8)	$c_i \to ld_{j,k}$	$\mathsf{Sign}_{sk(AS)}(c, pk(c_i))$
(9)	$ld_{j,k} \to c_i$	$Cert(ld_{j,k}), \mathsf{Enc}_{pk(AS)}(k'),$ $\mathsf{Enc}_{k'}(\mathsf{Open}(c, \mathsf{H}(n, id_s, T, pk(c_i), sv(ld_{j,k}))))$
(10)	$c_i \to AS$	$o, Cert(ld_{j,k}), \mathsf{Enc}_{pk(AS)}(k'),$ $\mathsf{Enc}_{k'}(\mathsf{Open}(c, \mathsf{H}(n, id_s, T, pk(c_i), sv(ld_{j,k}))))$
(11)	$AS \to c_i$	$\mathsf{Enc}_k(o)$

object to the trusted client, whose integrity is assumed to be intact (as confirmed using remote attestation). Finally, the use of multifactor authentication strengthens the identification of the requester.

The Read protocol could be extended in a straightforward manner to support *write* privileges, as well. Steps (1) – (10) are the same as above. In step (11), only $\mathsf{Enc}_{pk(c_i)}(k')$ is sent to c_i. In an additional step, c_i sends $\mathsf{Enc}_{k'}(o', \mathsf{Sign}_{sk(c_i)}(o'))$, where o' is the modified version of the object.

The next three protocols are used to enforce the continuity of usage constraints. Specifically, $\mathsf{Close}(\cdot)$ is used by the client (on behalf of the user) to end the session voluntarily. $\mathsf{Revoke}(\cdot)$ is initiated by the AS to terminate the session involuntarily, perhaps due to a violation of a proximity constraint. As the acknowledgment cannot be sent until the permission session has been terminated, a delay between these messages is to be expected. And $\mathsf{Renew}(\cdot)$ is executed during the session as a sort of "ping" protocol. That is, $\mathsf{Renew}(\cdot)$ is used to check that the connection to the AS is intact. In all three cases, c_i and the AS trivially exchange signed messages. As such, we will omit them for brevity.

Table 4.4
Protocol for movement $M(s, pa_i, pa_j)$

Move(s, pa_i, pa_j) – Subject s requests move from pa_i to pa_j.		
(1)	$s \rightarrow g_{i,j}$	Prove($sv(s)$)
(2)	$g_{i,j} \leftrightarrow g_{j,i}$	SyncEntry()
(3)	$g_{i,j} \rightarrow AS$	Commit(c,H($n, id_s, T, sv(g_{i,j})$))
(4)	$AS \rightarrow g_{i,j}$	Sign$_{sk(AS)}$(c)
(5)	$g_{i,j} \rightarrow AS$	Open(c,H($n, id_s, T, pk(c_i)$))
(6)	$AS \rightarrow g_{i,j}$	Sign$_{sk(AS)}$($grant(s, entry, open)$)
(7)	$g_{i,j} \leftrightarrow g_{j,i}$	OpenEntry()
(8)	$s \rightarrow g_{j,i}$	Prove($sv(s)$)
(9)	$g_{j,i} \leftrightarrow g_{i,j}$	LockEntry()
(10)	$g_{j,i} \rightarrow AS$	Commit(c',H($n', id_s, T', sv(g_{j,i})$))
(11)	$AS \rightarrow g_{j,i}$	Sign$_{sk(AS)}$(c')
(12)	$g_{j,i} \rightarrow AS$	Open(c',H($n', id_s, T', pk(c_i)$))

The final protocol, Move(s, pa_i, pa_j), is shown in Table 4.4 and is executed when a subject $s \in \mathcal{S}$ is moving from pa_i to pa_j. The protocol uses three additional primitives (SyncEntry(), OpenEntry(), and LockEntry()) that are executed between a synchronous, direct connection between the two guards $g_{i,j}$ and $g_{j,i}$ to control access to the entry between the PAs.

The protocol starts by s using his proximity device to prove knowledge of $sv(s)$. After claiming control of the door (SyncEntry()), $g_{i,j}$ sends a commitment c to the AS, who signs and returns the commitment. After confirming the signature, $g_{i,j}$ opens the commitment. If access is granted, the AS replies with a signed message indicating so. The door is unlocked, s passes through the entry, then uses his proximity device to connect with $g_{j,i}$ (indicating the move is complete). The door is then locked, and $g_{j,i}$ uses the commitment scheme to prove to the AS that the subject is now in pa_j.

Note that this protocol requires an honest s to re-authenticate with the guard on the other side of the entry. The purpose of these steps is to ensure that s passed through the entry. This portion of the protocol could be omitted if other techniques (*e.g.,* video monitoring or motion sensors) are used to guarantee such passage. Also, note that we are implicitly conflating the subject s and his proximity device. That is, we are not explicitly authenticating the person using the device. We find this approach acceptable, because, in our experience, institutions that are likely to deploy Prox-RBAC often strictly enforce badging procedures. That is, such groups currently require all individuals to swipe a badge at doors to protected areas, and use monitoring and auditing techniques to ensure employees adhere to these policies. If stronger assurances are desired, the protocol could be extended to require the user's password in addition to proving $sv(s)$.

4.4.3 Algorithms

In this section, we will describe the most important algorithms in our enforcement architecture that handle policy evaluation and movements. The algorithms themselves are listed in the appendix. We consider a number of primitive procedures (*e.g.,* role activation, Boolean evaluation, and the $WToWhileOnly$ function) to be very straightforward and efficient, and omit them for brevity. The first algorithm, **Request**, occurs between steps 10 and 11 of the Read protocol (or in the corresponding location of a Write protocol). This algorithm starts by calling **EvalPolicies**, explained below. If the request is approved, then the policies that are satisfied are checked for on-going conditions (*i.e., while* clauses). These policies are then added to $Ongoing(s)$, a list maintained by the AS that maps on-going conditions to object accesses for the subject s.

To expedite analysis of constraint satisfaction, we start the **EvalPolicies** algorithm by counting all of the instances of active roles. To ensure the hierarchy of PAs is captured, we traverse the $Parent$ function (defined by the partial order $\sqsubseteq$), mapping the roles accordingly. Note that we do not traverse the hierarchy defined by traditional RBAC roles, as we have not formally modeled this hierarchy. It is straightforward to add this traversal after line 8.

To complete the **EvalPolicies** algorithm, we must look at the process to determine if a constraint has been satisfied. The **Eval** algorithm handles this. The first part of

Algorithm 2: Request

```
Input: $s$ : the requesting subject ; $o$ : the requested object ; $c_i$ : the client ; $a$ : the
       requested action
Output: $approved$ or $denied$
if $Activate(s, r, Loc(s)) = failure$ then
    return $denied$ ;
$pol \leftarrow Policies[o][a]$ ;
$pol_{pre} \leftarrow EvalPolicies(s, pol)$ ;
if $pol_{pre} = \emptyset$ then
    return $denied$ ;
$pol_{on} \leftarrow \emptyset$ ;
foreach $p =< sr, a, o, \varphi > \in pol_{pre}, \varphi =< w, t >$ do
    $whileFound \leftarrow false$ ;
    if $w = \perp$ then
        for $i = 0$ to $n$ do
            /* $w =< w_0, b_1, w_1, \ldots, b_n, w_n >$ */
            if $w_0 \in \mathcal{W}_{while}$ then
                $whileFound \leftarrow true$ ;
        if $whileFound = true$ then
            $pol_{on} = pol_{on} \cup WToWhileOnly(p)$ ;
$Ongoing(s) = Ongoing(s) \cup \{< o, c_i, pol_{on} >\}$ ;
return $approve$ ;
```

the algorithm checks to see if the constraint uses the *this* keyword. If so, the parent tree will be traversed upward, starting at the subject's PA, until the referent PA is found (*i.e.,* this PA has a type that matches the constraint). Next, we find the list of PAs that can satisfy the constraint. $ParentSubtree(p)$ returns the subtree of the parent tree rooted at p, while $Neighbors(p2v(p))$ returns the list of PAs that share an entry with p. That is, $pa_j \in Neighbors(p2v(p))$ if and only if there is an edge $(p2v(pa_j), p2v(p)) \in E$. Finally, we count the number of role instances in the list of PAs, and compare the result with the constraint.

To analyze the efficiency of these algorithms, we first observe that $|paList| = \mathcal{P} \geq |\tau|$ in the worst case. So **Eval** is $O(n)$, where $n = |\mathcal{P}|$. To analyze **EvalPolicies**, we can assume $|\Phi| > |\mathcal{S}|$. Furthermore, let us assume that there is at least one *when* or *while* constraint for every role. Given these assumptions, we can see that the execution time of **EvalPolicies** is dominated by lines 9 and onward. Now, let

Algorithm 3: EvalPolicies

```
Input: $s$ : the requesting subject ; $P(o,a)$ : the set of matching policies
Output: $satisfied$ : the set of policies that have been satisfied
foreach $s' \in \mathcal{S}$ do
    if $s = s'$ then
        foreach $role =< r, pa >$ in $ActiveRoles(s')$ do
            $pa' \leftarrow pa$ ;
            repeat
                $count[pa'][r] \leftarrow count[pa'][r] + 1$ ;
                $pa' \leftarrow Parent(pa')$ ;
            until $pa' = \top$ ;
foreach $p =< sr, a, o, \varphi > \in P(o,a), sr =< r_p, pa_p >, \varphi =< w, t >$ do
    $permitted \leftarrow false$ ;
    foreach $role =< r_s, pa_s >$ in $ActiveRoles(s)$ do
        if $r_s \geq r_p$ and $p2v(pa_s) \sqsubseteq p2v(pa_p)$ then
            $permitted \leftarrow true$ ;
    if $permitted = true$ then
        if $w = \perp$ then
            $satisfied \leftarrow satisfied \cup \{p\}$ ;
        else
            for $i = 0$ to $n$ do
                for $j = 0$ to $m$ do
                    $csat_j \leftarrow Eval(c_j, count, Loc(s))$ ;
                $wsat_i \leftarrow BoolEval(csat_0, bc_1, csat_1, \ldots, bc_m, csat_m)$ ;
            if $BoolEval(wsat_0, bw_1, wsat_1, \ldots, bw_m, wsat_m)$ then
                $satisfied \leftarrow satisfied \cup \{p\}$ ;
return $satisfied$ ;
```

$n = max\{|\Phi|, n', m, |\mathcal{P}|\}$, where n' is the maximum number of *when* or *while* constraints in all policies and m is the maximum number of proximity constraints. Then **EvalPolicies** has a worst case execution time of $O(n^4)$. It is important to stress, though, that this is a theoretical upper bound. In practice, n' and m will both be small to be manageable. As such, in practical deployments, we can replace these values with a constant. Consequently, the realistic complexity would be $O(n^2)$.

If a policy is satisfied for a permission, and the policy contains at least one *while* constraint, the policy is added to $Ongoing(s)$ with all *when* constraints removed. At a regular interval (defined by the system designers), the AS will traverse this list and

Algorithm 4: Eval

```
Input: $c$ : the constraint to be satisfied ; $count$ : the $m \times n$ matrix representing the
        number of roles active in each PA ; $pa_s$ : the subject's location
Output: $true$ or $false$
/* $c = < c.q, c.n, c.r, c.r_t, c.p >$, where $c.q \in \{at\_least, at\_most, \epsilon\}$,
    $c.n \in \mathbb{N}_0, c.r \in \mathcal{R}, c.r_t \in \mathcal{R}_T, c.p \in \mathcal{P}$    */
$comp \leftarrow Quantifier2Inequality(c.q)$ ;
$total \leftarrow 0$ ;
if $c.p = this.\tau$ then
    $p \leftarrow pa_s$ ;
    while $Type(p) = \tau$ do
        $p \leftarrow Parent(p)$ ;
else
    $p \leftarrow c.p$ ;
if $c.r_t = in$ then
    $paList \leftarrow ParentSubtree(p)$ ;
else if $c.r_t = out$ then
    $paList \leftarrow \mathcal{P} - ParentSubtree(p)$ ;
else
    /* $c.r_t = adj$    */
    $paList \leftarrow Neighbors(p2v(p))$ ;
foreach $p'$ in $paList$ do
    $total \leftarrow total + count[p'][c.r]$ ;
return $total\ \ comp\ \ c.n$ ;
```

check that the conditions continue to hold. Note that, as long as *any* of the policies are satisfied, the subject retains access. Once all conditions have been violated, the *AS* will traverse the policy list for that object and initiate the Revoke protocol with the associated client. The timeout that is sent will be the *minimum* of the timeouts specified by the on-going policies. At the end of the algorithm, only the policies that are still satisfied are retained in $Ongoing(s)$.

There are two algorithms for handling the movement $M(s, v_i, v_j)$. The first, **Move**$_{5-6}$, occurs between steps 5 and 6 of the Move protocol. We start by checking if the subject is authorized (which can be specified using RBAC or identity-based policies). We then update the subject's roles to be linked to the least common parent of both pa_j and the PA associated with the role. Once the $ActiveRoles(s)$[3] list is

[3]Astute readers may note that $ActiveRoles(s)$ in our algorithms returns spatial roles $< r, pa >$, whereas $ActiveRoles(s)$ in our semantics returns only the traditional RBAC role r. This incompat-

updated, the on-going checks are re-evaluated, and the system marks that s has a pending move (which blocks further activations and requests). The second protocol **Move**$_{12}$ occurs after step 12 of the protocol. It simply sets $Loc(s)$ to be pa_j, then updates $ActiveRoles(s)$ in a manner similar to **Move**$_{5-6}$ by replacing the common parent with pa_j.

Algorithm 5: Move$_{5-6}$

```
Input: $s$ : the subject moving ; $pa_i$ : the old PA ; $pa_j$ : the new PA
Output: permission to pass through the entry or $\epsilon$
if $Authorized(s, pa_j) = false$ then
    return $\epsilon$ ;
$roles \leftarrow \emptyset$ ;
foreach $role =< r, pa >$ in $ActiveRoles(s)$ do
    if $pa_j \not\subseteq pa$ then
        $pa' \leftarrow LeastCommonParent(pa, pa_j)$ ;
        $roles \leftarrow roles \cup \{< r, pa' >\}$ ;
    else
        $roles \leftarrow roles \cup role$ ;
$ActiveRoles(s) \leftarrow roles$ ;
$CheckOngoing(s)$ ;
$PendingMove(s, pa_i, pa_j)$ ;
return $grant(s, e_{i,j}, open)$ ;
```

4.4.4 Functional Correctness

To illustrate the correctness of our work, we will offer two basic proofs that highlight the most important characteristics. First, we will prove that our algorithms satisfy the $< role, act, obj, when >$ semantics as a UCON$_{\text{preC}_0}$ policy. From there, the proofs for *while* policies are similar. Next, we will prove that our architecture correctly handles the movement $M(s, pa_i, pa_j)$.

ibility is trivial, but allowed us to specify the semantics more cleanly. Furthermore, the semantic definitions still captured the binding of roles to PAs in a way that matches the algorithms.

Algorithm 6: CheckOngoing

```
Input: $s$ : the subject with the permissions
$pol_{on} \leftarrow \emptyset$ ;
foreach $on =< o, c_i, pol_{pre} >$ in $Ongoing(s)$ do
    $pol_{post} \leftarrow EvalPolicies(s, pol_{pre})$ ;
    if $pol_{post} = \emptyset$ then
        $timeout \leftarrow \infty$ ;
        foreach $p =< sr, a, o, \varphi > \in pol_{pre}, \varphi =< w, t >$ do
            if $t = \epsilon$ then
                $timeout = 0$ ;
            else if $t < timeout$ then
                $timeout = t$ ;
        $Revoke(o, c_i, timeout)$ ;
    else
        $pol_{on} \leftarrow pol_{on} \cup \{< o, c_i, pol_{post} >\}$ ;
```

Lemma 1. The architecture enforces $< role, act, obj, when >$ policies as defined by the $\mathrm{UCON}_{\mathrm{preC}_0}$ semantics.

Proof. Assume **Request**(s, o, c_i, a, r) returns *approve*. Note that this can only occur at the end of the **Read** protocol, which ensures that the subject's identity and location are correct. By lines 1 and 2 of the algorithm, the role activation succeeds. Next, the request is approved only if lines 3 and 4 return a non-empty set of policies that are satisfied. By exploring **EvalPolicies**, we see that policies are only satisfied if there is an active role that is authorized (lines 11–13), and the specified constraints are met (lines 14–23). Clearly, the constraint evaluation corresponds to $preConChecked(\cdot)$, while the active role check corresponds to the remainder of the consequence of the implication. Thus, the consequence is true, and the implication holds. Therefore, the semantics are enforced correctly. $\square$

Lemma 2. The architecture correctly handles $M(s, pa_i, pa_j)$ events.

Proof. Recall that $M(s, pa_i, pa_j)$ requires authentication of the subject, authorization to pass through a physical barrier (*e.g.,* to open a locked door), and confirmation that the subject entered the new PA. Authentication occurs in steps 1–5 of the Move protocol and receiving authorization occurs in line 13 of the $\mathbf{Move}_{5-6}$ algorithm. Lines 6–12 of the Move protocol and the entirety of the $\mathbf{Move}_{12}$ algorithm combine to provide confirmation. □

4.5 Prototype Implementation

We have developed a proof-of-concept prototype of Prox-RBAC to measure the performance of the cryptographic protocols and the enforcement algorithms. To instantiave the Prove construct, we employed the Feige-Fiat-Shamir identification protocol [41], which uses a zero-knowledge proof, and we use a Pedersen commitment [121] for the Commit and Open primitives. For Auth, we simply used a salted hash of a password. We used SHA-256, AES-256, 1024-bit RSA, and SHA-1 with DSA for the Hash, Enc_k, $\mathsf{Enc}_{pk(c)}$, and Sign_k primitives, respectively. We implemented our prototype in Java 1.6.0_20, relying on standard cryptographic implementations when possible. For the Pedersen commitment and the Feige-Fiat-Shamir protocols, we used a custom implementation that employed the BigInteger class. Our test machine consisted of a 2.26GHz Intel® Core$^{\text{TM}}$ 2 Duo CPU with 3GB of 667MHz memory, running on Ubuntu 10.04 (“Lucid Lynx”) with version 2.6.32 of the Linux kernel. Based on 500 iterations, the most expensive of the cryptographic operations are the Pedersen commitment (average of 17.7 ms to generate and 20.2 ms to confirm) and RSA (5.7 ms to encrypt, but 30.2 ms to decrypt). Other than the DSA signature (9.8 ms average), all other computations required less than 1 ms on average to complete. The average time for the complete Read protocol (including the policy evaluation algorithms) was approximately 89.4 ms.

Moving toward a practical deployment with location sensing is more challenging. We have performed preliminary work toward using an Advanced Card Systems (ACS) NFC reader, model ACR 122 [110] to communicate with a Nokia 6131 NFC-enabled cell phone [108, 109]. Communication between the ACR 122 and the Nokia 6131 uses the peer-to-peer extension to the Java JSR 257 Contactless Communication API [111]. Our software employs the NFCIP library developed by Kooman [112], which uses the Java smartcardio libraries. One difficulty we had with this implementation is that the BigInteger class does not exist in the Java ME distribution. Consequently, deploying a protocol such as the Feige-Fiat-Shamir scheme requires developing one's own solution for large integers. However, based on our experiments, we observe that the average computation time for generating the Feige-Fiat-Shamir proof is less than the amount of time to perform the AES encryption in the protocol. Thus, we find such a deployment to be feasible.

4.6 Security Analysis

We can perform a security analysis of our architecture and implementation by focusing on two aspects. First, we can analyze attacks on the protocols and algorithms that originate from a malicious principal or from external adversaries; we present an informal analysis, including a description of the assumed motivation and abilities of the attacker. Second, we can analyze attacks that arise from our prototype implementation design choices.

In analyzing our protocols, we start with a corrupt subject. Such an adversary is essentially limited to denial-of-service attacks, barring stolen credentials. Now, if we consider a corrupt client[4], denial-of-service is trivial, because it can simply refuse to process requests. For threats to confidentiality, the client can store the user's password or it can hoard a commitment/open from the location device. However, the former is of limited use, as the client cannot initiate a request for a commitment from the

[4]Recall that we assumed trusted computing technologies were in place to ensure the client is not corrupt. However, this analysis assumes these technologies are not used.

location device; furthermore, future commitments are bound to the user's identity. In the latter case, storing the commitment/open is of no use, as the commitment scheme incorporates a timestamp and is assumed to be computationally binding (*i.e.,* the client, which has limited computational resources, cannot extract useful information). Finally, regarding integrity threats, all information that passes through the client is either encrypted or signed, thus ensuring that illicit modifications will be detected.

The biggest threat to our implementation is the use of the portable cell phones as the proximity device. Specifically, we chose to rely solely on the secret stored on the phone for authentication in the Move protocol. However, note that our particular cell phone, the Nokia NFC 6131, has a secure element that can protect sensitive information. Consequently, the user may be required to enter a PIN before accessing the secure element before initiating the Move. Observe also that known attacks on NFC and RFID technologies are not applicable. That is, our implementation uses the NFC peer-to-peer mode, which requires active computation by both of the devices. Thus, attacks that target secrets stored on passive RFID devices inherently fail, as there are no such persistent secrets that are revelaed in the clear in our protocols.

4.7 Conclusions

In this chapter, we have extended the notion of spatially aware RBAC to incorporate proximity constraints, which specify policy requirements that are based on the locations of other users in the environment. We have introduced our spatial model, primarily consisting of an accessibility graph that is based on existing work on graph-based indoor space models. We have also defined the syntax and semantics for the Prox-RBAC language for specifying these constraints. In addition to the formalization of our model and language, we have defined an enforcement architecture, including protocols and algorithms. We have offered preliminary results that prove the architecture correctly meets the semantic definitions. We have also described our initial work toward developing a prototype Prox-RBAC system, and closed with an informal security analysis. Based on these results, we find that it is feasible to construct a usable and efficient proximity-based RBAC system.

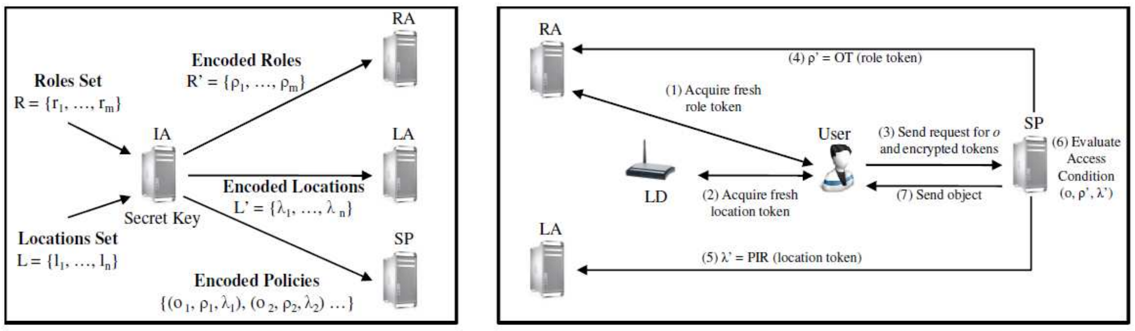
5 PRIVACY-PRESERVING ENFORCEMENT OF SPATIALLY AWARE RBAC

Up to this point, our threat model for spatially aware RBAC implicitly trusted the back-end service providers, and our primary focus for enforcement was authentication of the user and the user's environment. In this chapter, we consider a different challenge within the same realm of location-based access control. Specifically, we now consider how to protect the user's privacy from threats originating from the service providers. Given that our goal for this chapter is to ensure both privacy and security, which can often be conflicting goals, our discussion here will be significantly more rigorous and formal in order to make a clear delineation of the security guarantees of our design. Consequently, this chapter shows that it is possible to reconcile these goals.

5.1 User Privacy in the Organization

As we have already seen, GEO-RBAC has a large number of applications, in both military and civilian applications. Consider a military application, where physical presence in a secured room is required for a principal to access a confidential document. Such a protection model can prevent military personnel from unintentionally exposing secrets in an environment where principals without the appropriate clearance are present. Alternatively, a policy may state that strictly confidential documents must only be accessed in a room that has been checked thoroughly to ensure there are no unauthorized surveillance devices present. GEO-RBAC addresses these access control needs and supports permission manipulation at a fine granularity.

Civilian applications also benefit from authorization models and enforcement systems for location-based access control. For instance, medical personnel are often provided with mobile devices that allow them to access medical records of pa-


RA
Encoded Roles
R' = {ρ1, …, ρm}
Roles Set
R = {r1, …, rm}
IA
LA
Encoded Locations
L' = {λ1, …, λn}
Secret Key
Locations Set
L = {l1, …, ln}
SP
Encoded Policies
{(o1, ρ1, λ1), (o2, ρ2, λ2) …}
RA
(4) ρ' = OT (role token)
(1) Acquire fresh role token
User
(3) Send request for o and encrypted tokens
SP
(6) Evaluate Access Condition (o, ρ', λ')
LD
(2) Acquire fresh location token
(7) Send object
LA
(5) λ' = PIR (location token)

(a) Off-line Setup Phase (b) On-line Credential Retrieval and Access Control Enforcement Phase

Figure 5.1. Overview of privacy-preserving GEO-RBAC

tients.However, these devices may be stolen by a malicious adversary who may be able to break the device password and access the records of high-profile patients (*e.g.,* politicians, celebrities). Furthermore, some employees may not be trustworthy, and may attempt to access the record of such patients in a fraudulent manner. Enforcing that records can only be accessed within the perimeter of the healthcare provider, where physical surveillance and auditing mechanisms are in place to prevent unauthorized access, can prevent such detrimental leaks of sensitive medical records.

While the challenge of creating GEO-RBAC enforcement focuses primarily on authenticating the user's credentials and location claim, it would be desirable not to require disclosure of the requesting user's logical location or physical coordinates. Previous work [36] has acknowledged the severe privacy threats that may occur as a result of disclosing fine-grained location information with high frequency. This problem becomes even more serious in a decentralized, loosely coupled environment, such as cloud computing, where services (*e.g.,* data storage) are outsourced to an external provider. Thus, it is no longer the case that all components involved in the authorization process can be fully trusted to protect the privacy of the principals. For instance, private companies compete today for services of hosting data from healthcare providers. Even though authorization and auditing mechanisms may prevent unauthorized disclosure of sensitive records or to reduce the impact of a leak, similar protection mechanisms for the privacy of the requesting principals' locations do not exist.

Even when a centralized authorization infrastructure exists, and all GEO-RBAC components reside within the administrative control of a single organization, it is possible that a rogue administrator collects and uses principals' location for malicious purposes (*e.g.,* stalking, blackmail). Furthermore, the presence of malware at the service provider can also be a threat to users. Therefore, we argue that privacy-preserving access control enforcement is an important desideratum that should be supported by future GEO-RBAC systems.

In this chapter, we propose a framework for privacy-preserving GEO-RBAC (Priv-GEO-RBAC) that allows enforcement of location-based authorization without the need for the policy enforcement point (PEP) to learn the identity attributes or physical coordinates of the requesting users. Our approach relies on a combination of cryptographic techniques and separation of functionality among several distinct components. The trust assumption in our model is significantly reduced to one single component, which generates all cryptographic secrets required for evaluation. This trusted component *does not participate* in the on-line operations of the PEP, and therefore can be effectively shielded from attacks. The protocols we define ensure that the other components that participate in the on-line enforcement cannot learn the user's identity, role, or location, even under very powerful malicious assumptions. Our work shows that it is possible to reconcile security and privacy. Our approach supports fine-grained access control based on roles and spatial location while at the same time preserving the privacy of users whose accesses are enforced.

There are clear parallels between our work and access control mechanisms built on attribute-based encryption [122,123]. That is, location and role are used as attributes for policy evaluation. However, existing attribute-based schemes are built on the premise that attributes (*e.g.,* a driver's license number or date of birth) are fairly persistent for a user, and the user simply needs a credential that certifies the fact. In GEO-RBAC, roles and locations are inherently transient. While one user may activate a role for a short time, he will eventually be replaced by another user; additionally, users are inherently assumed to be mobile. Given the high frequency of attribute changes, temporary credentials that are dynamically and automatically generated are mandatory for usability. We refer to this problem as *attribute-based access control with* ***transient credentials***.

Figure 5.1 gives an overview of the proposed approach: in an off-line phase (Fig. 5.1(a)), executed only when the policies or the set of roles change, a trusted *Identity Authority (IA)* takes as input the set of all roles and locations (as we discuss later in Section 5.2, we use a discrete location space model) and encodes them using a secret mapping that never leaves the *IA*. The encoded roles locations and policies are then delivered to the *Role Authority (RA)*, *Location Authority (LA)* and *Service Provider (SP)*, respectively. The SP stores the protected objects and ultimately makes the decision of whether to grant access or not. Based on the encoded values, the *RA*, *LA* and *SP* are not able to link real roles or locations with the access request.

To ensure freshness of user credentials (thus preventing users from caching tokens for former locations) and to prevent replay attacks, we incorporate a time-based code generation approach that causes the credentials to expire. In the on-line phase (Fig. 5.1(b)), the user initiates a role session with the *RA*, acquiring a role token, then presents the role token to a *Location Device (LD)*, acquiring a location token that is bound to the role token (steps 1 and 2). The *LD* could be an RF-based sensor or a card reader present at the entrance to a room. The user then sends the credentials (which do not reveal the role and location) to the *SP* (step 3), who subsequently performs an oblivious transfer and a private information retrieval[1] with the *RA* and *LA* respectively, thus retrieving data to evaluate the access control policy (steps 4-5). The private retrieval steps are necessary in order to protect the access pattern of the principals: the *RA* and the *LA* do not learn anything about the encrypted credentials that are being used in the current access. In step 6 the *SP* evaluates the access

[1]We provide an overview of the OT and PIR protocols in Section 5.2

condition on *ciphertext credentials only*, and if the condition is satisfied, it grants access in step 7.

5.2 Background Material

As in the previous chapters, Priv-GEO-RBAC is built on the foundation of GEO-RBAC and UCON$_{ABC}$. As we have already discussed these models, we will omit repetition of these topics here. In addition, our space model is a simplified version as that defined in the previous chapter, and we will simply include a brief summary of it here. However, our cryptographic protocols depend on private information retrieval (PIR) and oblivious transfer (OT), and we provide an overview of these topics to provide the necessary background for the remainder of this chapter.

5.2.1 Space Model Summary

As described in the previous chapter, we rely on a partitioned reference space, as shown in Figure 4.3 in the previous chapter. That is, we assume policies are defined on a bounded and discrete geographical space with non-overlapping, hierarchical spatial regions. Furthermore, we assume that the user's movement between distinct areas is controlled, and the user must present physical access credentials (*e.g.,* a magnetic card) to a device that guards the entries between the protected areas. In addition, to ensure freshness of reported user locations, we employ a time-slice-based expiration mechanism of credentials, that dictates that the user must re-acquire a token from the *LD* in the enclosing area at a regular time interval. In practice, this is implemented by having the *LDs* and the *SP* share pseudo-random sequence of codes, with the help of a common random number generator seed. Combined with the forced re-authentication upon exiting, the expiration mechanism ensures that the user's location and path are constantly available for access control enforcement.

5.2.2 PIR and OT

Both Private Information Retrieval (PIR) and Oblivious Transfer (OT) protocols allow a user to fetch the value of a data item i from a server that owns an ordered set of data items $x_1 \ldots x_n$ of length n, without the server learning the value of i. However, they have some different requirements with respect to the communication complexity of the protocol, as well as the number of redundant items retrieved.

For simplicity, we consider the case where each data item is a bit, and hence the dataset is a bit string. A trivial solution is for the user to download the entire bit-string, but such an approach has communication cost $\Theta(n)$. A *non-trivial* PIR protocol has two requirements: (P1) protect user privacy, *i.e.,* keep the value of i secret from the server, and (P2) incur sub-linear (*i.e.,* $o(n)$) communication cost. Chor *et al.* [124, 125] were the first to introduce the PIR concept, and they showed that in an information-theoretic setting, there is no non-trivial solution with a single server. The same authors propose a scheme with $K \geq 4$ non-colluding servers with $O(n^{1/\log K} K \log K)$ communication cost. This bound was improved to $O(n^{\frac{\log \log K}{K \log K}})$ in [126]. More recently, Yekhanin [127] showed that if infinitely many Mersenne primes exist, then there is a three-server PIR protocol with $O(n^{1/\log \log n})$ communication complexity.

In practice, the assumption of multiple non-colluding servers may not often be met. Computational PIR (cPIR) is concerned with finding efficient single-server solutions that are robust against adversaries with polynomially-bounded computation capabilities. The seminal work of Kushilevitz and Ostrovski [128] proposed the first cPIR protocol that relies on the Quadratic Residuosity Assumption (QRA) and has $O(n^{\epsilon})$ communication cost for arbitrarily small $\epsilon > 0$. The bound was improved by Cachin *et al.* [129] who proposed a poly-logarithmic communication protocol that relies on the ϕ-hiding assumption (ϕHA). More efficient solutions have been proposed for a slight variation of PIR, namely Private Block Retrieval (PBR). In PBR, the user retrieves a block of bits, rather than a single bit. Lipmaa [130] introduced a $O(\log^2 n)$

communication protocol, whereas Chang [131] proposed a solution with $O(\log n)$ communication complexity based on the Paillier [132] cryptosystem. Finally, Gentry *et al.* [133] introduced a constant-rate, communication-efficient PBR protocol that can be used under several distinct intractability assumptions, including ϕHA.

OT is a closely-related concept to PIR, and was originally introduced in [134]. OT, also referred to as *1-out-of-n* transfer, has two requirements: (O1) the server should not learn the value of i, and (O2) the client should not learn the value of any bit other than the i^{th} bit. Note that, OT does not specify any condition on efficiency, as opposed to requirement (P2) of PIR. In fact, several OT protocols, e.g., [135], have $\Theta(n)$ communication complexity. The underlying idea behind OT is to encrypt each data item with a different encryption key, for a total of n keys. Then, the client is allowed to retrieve bit-by-bit the encryption key of element i, using for each bit of the key a *1-out-of-*2 protocol that works as follows: assume that the server has two messages m_0 and m_1, and the client has a bit $b \in \{0, 1\}$. The client wishes to learn message m_b without letting the server find b, whereas the server wants to ensure that the client only learns one of the two messages. The following steps are performed:

1. The server chooses a public-key encryption algorithm (E, D), where the ciphertext space is the entire space of m-bit values, and two random m-bit values x_0 and x_1. Then it sends the client E, x_0 and x_1.

2. The client chooses a random value c and sends $q = E(c) \oplus x_b$ to the server.

3. The server computes $c_0 = D(q \oplus x_0)$, $c_1 = D(q \oplus x_1)$, $d_0 = m_0 \oplus c_0$, $d_1 = m_1 \oplus c_1$, and sends d_0, d_1 to the client.

4. The client computes $m_b = d_b \oplus c$.

At the end of the protocol, the user is able to decrypt only the data item at position i in the set of items.

5.3 Proposed Framework

5.3.1 Preliminaries

- $\mathcal{R}$ denotes the set of traditional RBAC roles.
- $\mathcal{L}$ denotes the set of locations defined by a protected area PA.
- $\mathcal{O}$ denotes the set of objects to be protected.
- $\mathbb{A}$ denotes the set of actions that can be performed on objects.
- $\mathcal{S}$ denotes the set of subjects that can make requests.

Our assumption is that $|\mathbb{A}| \ll |\mathcal{R}| \ll |\mathcal{L}|$, implying that there are very few possible actions, a moderate number of roles, and many locations. As $|\mathbb{A}|$ and $|\mathcal{S}|$ do not directly impact our protocol design, we place no assumptions on the size of these sets.

5.3.2 Principals

Let $\mathcal{P}$ denote the set of all possible principals, defined as follows.

- **Client (*C*)** – the principal representing the subject making the request. For each such C, there is a unique subject $s \in \mathcal{S}$, but not necessarily vice versa. For instance, the same user may simultaneously initiate sessions with multiple devices (*e.g.*, a smartphone and a laptop); we treat these sessions independently as multiple clients. In our framework, we will refer to the client's public key $pk(C)$, which could be linked to either the user or the device according to the needs of the deployed system.[2]

[2]Observe that using the same key for multiple sessions (*e.g.,* if the user creates sessions on multiple devices with the same public key), this could be a privacy concern, as these sessions are linkable. A straightforward solution would be to prohibit users from using the same key for multiple sessions.

- **Location device (*LD*)** – a small, proximity-based embedded device that identifies a protected area. In practice, as protected areas may be large, we assume each location $l \in \mathcal{L}$ would have multiple such devices for ease-of-use. However, we treat these devices collectively as a single principal *LD*.

- **Role authority (*RA*)** – a centralized RBAC server that is tasked with authenticating users, as well as creating and maintaining session identifiers.

- **Location authority (*LA*)** – a centralized server that maintains information on the set of *LD*s.

- **Service provider (*SP*)** – a server that controls access to a protected resource. We assume that there are multiple such servers, and the client knows which *SP* to contact when retrieving a desired service.

- **Identity authority (*IA*)** – a trusted third-party server that establishes and maintains information on the identities of all users and location devices. *IA* also is responsible for encoding policies for each *SP*, however it does not maintain this information after the setup phase of our framework. Moreover, *IA* is not involved in any of the run-time protocols, and only exists as a trusted entity for establishing identity credentials.

5.3.3 Cryptographic Primitives & Notation

Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ denote an encryption scheme that provides indistinguishable encryptions under chosen plaintext attacks (IND-CPA-secure)[3] such that $\mathsf{Gen}(1^n)$ denotes a probabilistic key generation algorithm, $\mathsf{Enc}_k(\cdot)$ denotes encryption using the key k, while $\mathsf{Dec}_k(\cdot)$ denotes the corresponding decryption routine. As key generation

[3]We stress here the importance that encryption must be IND-CPA-secure. Specifically, IND-CPA provides a probabilistic guarantee that encrypting the same message with the same key multiple times produces *distinct* ciphertexts. Consequently, given $c_1 \leftarrow \mathsf{Enc}_k(m_1)$ and $c_2 \leftarrow \mathsf{Enc}_k(m_2)$, an observer cannot determine $m_1 \overset{?}{=} m_2$ without decrypting the messages, even if the encryption key k is known. See [136] for further discussion.

and encryption are probabilistic, we adopt the standard convention to denote whether or not assignment is deterministic:

$$k \leftarrow \mathsf{Gen}(1^n) \qquad\qquad c \leftarrow \mathsf{Enc}_k(m) \qquad\qquad m := \mathsf{Dec}_k(c)$$

Our convention is to use the same notation, regardless of whether symmetric or public key encryption is used, as the context clearly identifies which is needed. We denote the public key of principal p as $pk(p)$, whereas symmetric keys are written as $\mathcal{K}_i$. In the case of Gen, 1^n is a security parameter for the key generation. One final requirement of the public key encryption used in our scheme is that it must be commutative, thus allowing out-of-order decryptions. For instance, El Gamal [137] is one such scheme. Thus, given $c \leftarrow \mathsf{Enc}_{pk(A)}(\mathsf{Enc}_{pk(B)}(m))$,

$$m := \mathsf{Dec}_{pk(A)}(\mathsf{Dec}_{pk(B)}(c)) = \mathsf{Dec}_{pk(B)}(\mathsf{Dec}_{pk(A)}(c))$$

In addition to encryption, we denote a collision-resistant hash function as $\mathsf{H}(\cdot)$, and $\mathsf{Auth}(\cdot)$ denotes an interactive authentication protocol. The details of $\mathsf{Auth}(\cdot)$ are tangential to our scheme and may be selected as desired; the arguments indicate the entity to authenticate. We also deploy two privacy-preserving schemes, as described in Section 5.2. We denote oblivious transfer as $\mathsf{OT}(i)$, where i indicates the index of the record to be retrieved. Similarly, $\mathsf{PIR}(i, X(i))$ denotes private information retrieval for the index i and the items $X(i)$.

Our protocol relies on subtle facets of the RSA assumption that warrant explicit discussion. Recall that the RSA assumption states that, given $N = pq$ (p and q are large primes), e coprime with $\phi(N)$, and some y, it is intractable to find x such that $x^e \equiv y \mod N$. The critical point for our scheme is that this relies on the difficulty of computing multiplicative inverses modulo $\phi(N)$ when $\phi(N)$ is unknown. Otherwise, the RSA assumption would crumble, as one could efficiently compute d such that $ed \equiv 1 \mod \phi(N)$ (*i.e.*, $d = e^{-1 \mod \phi(N)}$) and $y^d = (x^e)^d = x^{ed} = x^{1 \mod \phi(N)} \equiv x \mod N$. Thus, the RSA assumption implies that *computing multiplicative inverses modulo $\phi(N)$ is intractable when the factorization of N is unknown.* This implication has a direct impact on our notation. Specifically, when we write something of the form $\iota^{x^{-1}}$, all operations in the exponent are evaluated modulo $\phi(N)$, while the exponentiation is performed modulo N. That is, x^{-1} is the multiplicative inverse of x modulo $\phi(N)$ (which requires knowledge of $\phi(N)$ to compute). To make the distinction clear, the following equivalence holds, and we generally omit the modular exponentiation notation:

$$\iota^{x^{-1} \bmod \phi(N)} = \iota^{x^{-1}} \mod N$$

5.3.4 Identity Establishment

Our role and location authentication scheme relies on cryptographic properties of the group $\mathbb{Z}_N^*$ such that the RSA assumption is satisfied.[4] The value N is made public to all principals of our framework. *IA* selects a number of unique role identifiers as follows. First, $\iota \in \mathbb{Z}_N^*$ serves as the basis of our identification scheme. Next, *IA* selects $\rho \in \mathbb{Z}_N^*$ such that $\sqrt{N} \leq \rho \leq \phi(N)$. *IA* provides *RA* with ι, ρ, and $\phi(N)$, so that *RA* can compute multiplicative inverses as needed. *IA* also ensures that *RA* knows the mapping from ρ to the corresponding RBAC role. These values must be kept secret to *RA*.

IA then creates location identifiers $\lambda \in \mathbb{Z}_N^*$ subject to the same constraint that $\sqrt{N} \leq \lambda \leq \phi(N)$. This size constraint ensures that $\rho\lambda \geq N$, which strengthens the security guarantees of our scheme. In crafting the λ values, we place one additional restriction: *IA* must ensure there are no colliding pairs of products $\rho\lambda$. That is, for $\rho, \widehat{\rho}, \lambda, \widehat{\lambda}$ with $\rho = \widehat{\rho}$ and $\lambda = \widehat{\lambda}$, *IA* ensures that $(\rho\lambda) = (\widehat{\rho\lambda})$. If such a collision occurs, *IA* discards one of the values and selects a new element from $\mathbb{Z}_N^*$. Once all values are created, *IA* provides *LA* with $\iota^{-1 \bmod \phi(N)}$, the λ values, and the mapping to the corresponding locations.

Finally, *IA* maintains a persistent mapping for actions to values $\alpha \in \mathbb{Z}_N^*$, also subject to the constraint $\sqrt{N} \leq \alpha \leq \phi(N)$. The mapping of α to actions is kept private to *IA*. Observe that the values ρ and λ do not have any particular sensitivity,

[4]Recall that $\mathbb{Z}_N^* = \{a \in \{1, \ldots, N-1\} \mid \gcd(a, N) = 1\}$. This ensures that all such $z \in \mathbb{Z}_N^*$ have unique multiplicative inverses.

unless the mappings from the values to the corresponding roles and locations are known. However, if *SP* knows these values, it provides linkability between policies and requests. In order to minimize this threat, one of the goals of our scheme is to reduce the likelihood that *SP* is able to form such correlations. We will explore this discussion more in Section 5.4.

5.3.5 Priv-GEO-RBAC Policies

In Priv-GEO-RBAC, for the object $o \in \mathcal{O}$, service providers define a single **object policy** P_o such that $P_o =< p_{o.1}, \ldots, p_{o.n} >$, where

$$p_{o.i} \;=\; < r, l, a > \text{ for some } r \in \mathcal{R}, l \in \mathcal{L}, a \in \mathbb{A}$$

The policy semantics are based on a white-list, meaning the presence of a policy $p_{o.i}$ grants the permission for role r to perform action a on o while in location l. Absence of such a policy indicates denial of the request. For notation, $p_{o.i}[r]$ denotes the role, $p_{o.i}[l]$ denotes the location, and $p_{o.i}[a]$ denotes the action.

When *SP* establishes or changes the policy for an object o, *SP* contacts *IA* to encode the policies. *IA* first creates a new object identifier $\delta \in \mathbb{Z}_N^*$ such that $\sqrt{N} \leq \delta \leq N$. As with $\rho\lambda$, *IA* ensures that, for all actions, there are no colliding products $\alpha\delta = \widehat{\alpha}\widehat{\delta}$ when $\alpha = \widehat{\alpha}$ or $\delta = \widehat{\delta}$. Now, consider the policy $p_{o.i} =< r, l, a >$. *IA* finds the identifiers $\rho, \lambda, \alpha \in \mathbb{Z}_N^*$ for the role, location, and action. The policy is then encoded as the tuple

$$\widehat{p_{o.i}} \;=\; < (\rho\lambda)^{-1}(\alpha\delta) \mod \phi(N), \iota^{\alpha\delta} \mod N >$$

where the $(\rho\lambda)^{-1}$ denotes the multiplicative inverse of $(\rho\lambda)$ modulo $\phi(N)$.

Note the following properties of this encoding. First, as there are no colliding products $\alpha\delta$, there will be no object-action pairs that have the same encoded policies, even if the role-location pairs are the same. This prevents *SP* from linking encoded policies. Next, without knowledge of $\phi(N)$, *SP* cannot compute the multiplicative inverse $((\rho\lambda)^{-1}(\alpha\delta))^{-1 \mod \phi(N)} = (\rho\lambda)(\alpha\delta)^{-1 \mod \phi(N)}$. If *SP* could compute this value, it would have:

$$(\iota^{\alpha\delta})^{(\rho\lambda)(\alpha\delta)^{-1}} \equiv \iota^{\rho\lambda} \mod N$$

which would clearly allow linkability for any encoded policy with the same role-location pair. However, *this calculation requires knowledge of $\phi(N)$, which* SP *lacks.* Finally, given the assumption that α and δ are large and *SP* has no knowledge of α, δ, or ι, *SP* cannot link policies across different objects or actions.

IA returns the encoded object policy $\widehat{P_o}$ that consists of the array $< \widehat{p_{o.1}}, \ldots, \widehat{p_{o.n}} >$ in a random order. This shuffling prevents *SP* from determining $\widehat{p_{o.i}} \overset{?}{\equiv} p_{o.i}$. Observe, though, that all encoded policies relating to the same action on the same object share the value $\iota^{\alpha\delta}$. Consequently, *SP* is able to group the encoded policies according to the action. Based on this grouping, we can refer to the policy group $\widehat{P_{o.a}} =< \widehat{p_{o.a.1}}, \ldots, \widehat{p_{o.a.m}} >$ for the object o and action a. Note that, as *SP* has no knowledge of ρ, λ, ι, or δ, it can only see the policies as distinct pairs of large integers.

5.3.6 Protocols

In this section, we define the protocols of our framework. In these definitions, we only explicitly identify encryptions that are required to prevent one of the legitimate principals from learning a protected value. For instance, Protocol 1 sends the symmetric key K_c in the clear, as both *RA* and *C* are authorized to know this key; on the other hand, Protocol $\mathcal{Q}$ sends multiple pieces of data encrypted with a public key to prevent one of the principals from learning the encrypted value. Consequently, system deployments should consider the security threats of the underlying communications channel and add cryptographic protections (including MACs) as needed.

Protocol for RBAC session creation. Figure 5.2 shows Protocol 1, which used to create a new RBAC session. After authenticating the user (and the user's authority to activate role $r \in \mathcal{R}$), *RA* generates a number of session parameters as follows. The value $x \in \mathbb{Z}_N^*$ is a random positive integer, b denotes a nonce used in the $\mathsf{Auth}(\cdot)$ scheme (known to both *RA* and *C*), and pwd_c is the user's password.[5]

[5]If the particular instantiation of $\mathsf{Auth}(\cdot)$ does not require a nonce b, *RA* can generate one for *C*.

Protocol 1 – creating a new role session

```
1) $[C \leftrightarrow RA]$ $\mathsf{Auth}(C, r, b)$
     $[RA]$ $\sigma_r := \; < \iota^{\gamma}(\iota^{\rho})^{x} = \iota^{\gamma}\iota^{\rho x}, x^{-1} \mod \phi(N) >$
     $[RA]$ $\mathsf{K}_r \leftarrow \mathsf{Gen}(1^n)$
     $[RA]$ $\mathsf{K}_c \leftarrow \mathsf{Gen}(1^n)$
     $[RA]$ $record := \mathsf{Enc}_{\mathsf{K}_r}(\sigma_r \parallel \mathsf{K}_c \parallel valid)$
     $[RA]$ $\tau_r \leftarrow \mathsf{Enc}_{pk(RA)}(i_r \parallel \mathsf{K}_r \parallel \mathsf{H}(b \parallel pwd_c))$
2) $[RA \rightarrow C]$ $\tau_r, \mathsf{K}_c$
```

Protocol 2 – retrieving a proof-of-location

```
1) $[C \rightarrow LD]$ $\tau_r$
     $[LD]$ $sig := \mathsf{H}(d \parallel code_l[T] \parallel \tau_r)$
     $[LD]$ $\tau_l \leftarrow \mathsf{Enc}_{pk(LA)}(i_l \parallel d \parallel exp \parallel T \parallel sig)$
2) $[LD \rightarrow C]$ $\tau_l$
```

Figure 5.2. Protocols for retrieving tokens τ_r and τ_l

RA creates a new *record* in its session database with a commitment token σ_r, and marks the record with a boolean value *valid*. Unless the system allows simultaneous activation of multiple roles, RA invalidates all other records for this user by flipping the corresponding boolean value in those records. We refer to τ_r as the **role token**. The rationale for encrypting the record with the key K_r is to ensure that the information can only be accessed by a SP with the corresponding role token. Alternative approaches [138] have been proposed to integrate access control with OT, but encrypting the record is sufficient for our framework. The other key K_c is generated to ensure the object can be accessed only by the legitimate C.

Astute readers will note the similarity in structure between $\iota^{\gamma}\iota^{\rho x}$ and the Pedersen commitment scheme [121]. In the Pedersen commitment, for a cyclic group generated by g, the prover aims to commit to $h = g^x$, where x is unknown. To do so, the prover first generates $g^s h^t$ for random values s and t. Later, the prover reveals s and t. This scheme is proven to be *perfectly hiding* of the exponent x. While our goals our different (*e.g.,* we do not require information theoretic security), this structure ensures that, for any two role sessions with the same ρ, $\iota^{\gamma}\iota^{\rho x} = \iota^{\gamma}\iota^{\rho \widehat{x}}$, when $x = \widehat{x}$.

Furthermore, the obfuscating factor ι^γ ensures that knowledge of x^{-1} and $\widehat{x^{-1}}$ are insufficient to determine if the ρ and $\widehat{\rho}$ values are equivalent in two distinct sessions.

Protocol for retriving a proof-of-location. When the user wishes to make a request, he must retrieve a proof-of-location from a *LD* for the corresponding protected area. This proof must bind the user session to the location at a particular time. As such, *LD* will generate the following data, where d denotes a nonce, exp is an expiration time, i_l is the index for *LD* in the location database, $code_l[T]$ is the value of a rolling passcode (*i.e.,* it repeatedly changes after a set time interval) at timestamp T, and τ_l is the **location token**. Additionally, note that the location token τ_l is dependent on the role token τ_r, thus binding the role session to the location. The protocol for retrieving the proof-of-location proceeds as follows.

Protocol for policy enforcement. Figure 5.3 shows Protocol $\mathcal{Q}$. Once *C* has the role and location tokens, *C* initiates Protocol $\mathcal{Q}$ with the relevant *SP* to request access to perform action $a \in \mathbb{A}$ on object $o \in \mathcal{O}$. For simplicity, we will assume the action is *read*.[6] *C* sends the tokens to *SP*. *SP* gets *RA* and *LA* to perform a blind decryption (*i.e., RA* and *LA* do not learn the decrypted messages). Using this information, *SP* authenticates *C* with a traditional password, retrieves role session information from *RA* using OT, and retrieves location information from *LA* using PIR. Finally, *SP* evaluates the encoded policies based on the data retrieved from *RA* and *LA*. The evaluation procedure ensures that *SP* only learns *if* a policy is satisfied; for a single request, *SP* cannot determine the user's identity, location, or role. Section 5.4 examines these properties in detail. In the following definition of $\mathcal{Q}$, z denotes a nonce, while $m \in \mathbb{Z}_N^*$ is selected at random from $\sqrt{N} \leq m \leq N-1$.

The first important point to emphasize in this protocol is the blinded decryption in step 3. That is, $\tau_r \leftarrow \mathsf{Enc}_{pk(RA)}(\cdot)$, so $e_{sr} \leftarrow \mathsf{Enc}_{pk(SP)}(\mathsf{Enc}_{pk(RA)}(\cdot))$. By decrypting this message blindly, *RA* is sending the following to *SP*:

[6]It is straightforward to add support for *write* actions by appending MACs as necessary. Note, though, that using a persistent secret key to generate the MAC will allow *SP* to link writes. Instead, τ_r should be augmented with a session public key, and Protocol 1 should send the corresponding session private key to *C*.

Protocol $\mathcal{Q}$ – requesting access to a protected resource

```
1)  $[C \to SP]\ \tau_r, \tau_l, o, a$
2)  $[SP \to RA]\ e_{sr} \leftarrow \mathsf{Enc}_{pk(SP)}(\tau_r)$
3)  $[RA \to SP]\ e_s := \mathsf{Dec}_{sk(RA)}(e_{sr})$
      $[SP]\ (i_r \parallel \mathsf{K}_r \parallel \mathsf{H}(b \parallel pwd_c)) := \mathsf{Dec}_{sk(SP)}(e_s)$
4)  $[SP \to C]\ z \in \{0,1\}^*$
5)  $[C \to SP]\ h := \mathsf{H}(z \parallel \mathsf{H}(b \parallel pwd_c))$
6)  $[SP \leftrightarrow RA]\ e_{OT} := \mathsf{OT}(i_r)$
      $[SP]\ (\sigma_r \parallel \mathsf{K}_c \parallel valid) := \mathsf{Dec}_{\mathsf{K}_r}(e_{OT})$
7)  $[SP \to LA]\ \tau_l$
8)  $[LA \to SP]\ e'_s := \mathsf{Dec}_{sk(LA)}(\tau_l)$
      $[SP]\ (i_l \parallel d \parallel exp \parallel T \parallel sig) := \mathsf{Dec}_{sk(SP)}(e'_s)$
9)  $[SP \leftrightarrow LA]\ (\lambda \parallel code_l[T]) := \mathsf{PIR}(i_l, X(i_l))$
10) $[SP \to RA]\ \iota^{-\lambda m}$
11) $[RA \to SP]\ (\iota^{-\lambda m})^{\gamma} = \iota^{-\lambda m \gamma}$
      $[SP]\ v := \mathsf{Eval}(\sigma_r, \lambda, \iota_{lmg}, exp, sig)$
12) $[SP \to C]\ e_o \leftarrow \mathsf{Enc}_{\mathsf{K}_c}([o])$
```

Figure 5.3. Access control enforcement protocol

$$\begin{aligned}
e_s &:= \mathsf{Dec}_{sk(RA)}(e_{sr}) \\
&= \mathsf{Dec}_{sk(RA)}(\mathsf{Enc}_{pk(SP)}(\mathsf{Enc}_{pk(RA)}(\cdot))) \\
&= \mathsf{Dec}_{sk(RA)}(\mathsf{Enc}_{pk(RA)}(\mathsf{Enc}_{pk(SP)}(\cdot))) \\
&= \mathsf{Enc}_{pk(SP)}(\cdot)
\end{aligned}$$

which can then be decrypted by *SP*. However, the IND-CPA encryption by *SP* ensures that *RA* cannot determine the original encrypted message, so *RA* fails to learn which role session is being used. Step 8 uses the same technique for locations.

The need for two separate privacy-preserving schemes, OT and PIR, may not be intuitive. This choice was deliberate, as the needs of the respective portions of the protocol are different. First, the role session information is more sensitive, as it includes the key K_c used to communicate with the user. As such, it is important that the *SP* only retrieves the records for that session. PIR cannot provide this guarantee. Second, the location database contains more information than is required by the protocol. Specifically, each record contains additional information about the protected area. PIR allows *SP* to retrieve only the relevant data. Also, the passcode

$code_l[T]$ is only good for a short period of time before it expires. As such, *SP* would have a very short window of opportunity to exploit knowledge of the passcode for other *LDs*. Furthermore, this leak is *not a security threat*, as we will show in our security analysis. Thus, PIR is the correct choice for retrieving this data from *LA*.

The evaluation procedure consists of checking that the location token has not expired (*i.e.,* current time is prior to *exp*), validating the location signature *sig*, and evaluating the policy set $P_{o.a}$. To validate *sig*, *SP* does a straightforward comparison:

$$sig \overset{?}{=} \mathsf{H}(d \parallel code_l[T] \parallel \tau_r)$$

To prevent network lag from preventing the code to match, a straightforward adaptation would be for $code_l[T]$ to denote multiple, consecutive codes. Evaluating the policy set requires performing a number of calculations. Given the importance of this evaluation to our protocol, we will describe and evaluate this procedure in detail in the following section.

5.3.7 Functional Correctness

In this section, we focus on the functional correctness of the policy evaluation. That is, we show that *SP* can evaluate the encoded policies correctly, given the role and location information encoded by σ_r and λ. Consider a request for object $o \in \mathcal{O}$, identified by $\delta \in \mathbb{Z}_N^*$, where the action $a \in \mathbb{A}$ is identified by α. Recall that this uniquely identifies the set $\widehat{P_{o.a}} =< \widehat{p_{o.a.1}}, \ldots, \widehat{p_{o.a.n}} >$. In addition, *SP* retrieved $\sigma_r :=< \iota^\gamma \iota^{\rho x}, x^{-1} \mod \phi(N) >$ from *RA* during the OT step of the protocol, and *SP* retrieved λ from *LA* during the PIR. Next, *SP* selected a nonce $m \in \mathbb{Z}_N^*$, and used λ and $\iota^{-1 \mod \phi(N)}$ to calculate $\iota^{-\lambda m} \mod N$. *SP* sent this value to *RA*, who responded with $\iota^{-\lambda m \gamma}$, where γ is the persistent obfuscator used by *RA*. As γ is large, it is intractable for *SP* to compute the discrete logarithm and learn γ. Similarly, *RA* cannot determine λ or m. Now, consider an encoded policy $\widehat{p_{o.a.i}} \in \widehat{P_{o.a}}$, which we denote as $< s, \iota^{\alpha\delta} >$. In order to evalute $\widehat{p_{o.a.i}}$, *SP* performs the following calculations:

1. $(\iota^{\gamma}\iota^{\rho x})^{\lambda m} = \iota^{\gamma\lambda m}\iota^{\rho x\lambda m}$

2. $\iota^{-\lambda m\gamma} \cdot \iota^{\gamma\lambda m}\iota^{\rho x\lambda m} \equiv \iota^{0+\rho x\lambda m \mod \phi(N)} = \iota^{\rho x\lambda m}$

3. $((\iota^{\rho x\lambda m})^{s})^{x^{-1}} = \iota^{\rho x\lambda m s x^{-1}} \equiv \iota^{\rho\lambda m s \mod \phi(N)}$

4. $\iota^{\rho\lambda m s} \stackrel{?}{=} (\iota^{\alpha\delta})^{m}$

The policy is satisfied if and only if the equality holds. For clarity in the following proofs, we refer to this test as the **policy equation**.

Theorem 1. Under the assumption that all parties behave honestly, the policy equation is satisfied if and only if the access control policy $p_{o.i}$ is satisfied.

Proof. There are two cases to consider. First, assume $p_{o.i}$ is satisfied, given the credentials used in the protocol. In that case, $s = (\rho\lambda)^{-1}(\alpha\delta)$ (*i.e.,* the role identified by ρ and the location identified by λ match those in the request). Observe that

$$\begin{aligned}
(\iota^{\rho\lambda m s})^{(\rho\lambda)^{-1}(\alpha\delta)} &= \iota^{\rho\lambda m(\rho\lambda)^{-1}(\alpha\delta)} \\
&= (\iota^{(\rho\lambda)(\rho\lambda)^{-1}})^{m\alpha\delta} \\
&\equiv (\iota^{1 \mod \phi(N)})^{m\alpha\delta} \\
&= \iota^{m\alpha\delta}
\end{aligned}$$

The equivalence follows from Euler's theorem, which states that $\iota^{\phi(N)} = 1$ if ι and N are coprime. Consequently, the policy equation holds:

$$\iota^{\rho\lambda m s} \equiv \iota^{m\alpha\delta} \mod N = (\iota^{\alpha\delta})^{m}$$

Therefore, if the credentials satisfy the original policy, then the policy equation holds. Now, consider the other implication. We must show that, if the policy equation holds, then the original policy is satisfied. We will proceed with a proof by contradiction. Assume the policy equation holds, but the credentials presented do *not* satisfy the policy. Since the policy is not satisfied, $s = (\widehat{\rho\lambda})^{-1}(\alpha\delta)$, where either $\widehat{\rho} = \rho$, $\widehat{\lambda} = \lambda$, or both. Assume $\widehat{\rho} = \rho$ but $\widehat{\lambda} = \lambda$. Then

$$(\rho\lambda)(\widehat{\rho}\lambda)^{-1} = (\rho\widehat{\rho}^{\,-1})(\lambda\lambda^{-1}) \equiv (\rho\widehat{\rho}^{\,-1}) \mod \phi(N)$$

Observe that the policy equation will only hold if $\iota^{\rho\widehat{\rho}^{\,-1}} = \iota$, which can only occur if $\rho\widehat{\rho}^{\,-1} \equiv 1 \mod \phi(N)$. However, this requires that $\widehat{\rho}^{\,-1}$ is the inverse of ρ, meaning $\widehat{\rho} = \rho$, which contradicts our assumption. Thus, if $\widehat{\lambda} = \lambda$ and the policy equation holds, then $\widehat{\rho} = \rho$. By the same rationale, if $\widehat{\rho} = \rho$ and the policy equation holds, then $\widehat{\lambda} = \lambda$. Thus, if the policy equation holds and one of the credentials is correct, then the other must be correct, contradicting the assumption that the policy is not satisfied. Now, assume $\widehat{\rho} = \rho$ and $\widehat{\lambda} = \lambda$. If the policy equation holds, then

$$(\rho\lambda)(\widehat{\rho\lambda})^{-1} \equiv 1 \mod \phi(N)$$

This implies $(\widehat{\rho\lambda})^{-1}$ is the inverse of $(\rho\lambda)$, implying $(\widehat{\rho\lambda}) = (\rho\lambda)$. However, we explicitly prohibited such a collision of products. That is, *IA* is prevented from creating such a pair. Thus, if the policy equation holds, then the credentials provided must be correct. $\square$

5.4 Security Analysis

In this section, we present an analysis of our protocol in two ways. First, we consider the security of our framework under the Dolev-Yao adversarial model [139]. Under this model, an adversary $\mathcal{A}$ is capable of sending and receiving messages, decrypting messages with known keys, storing data, and generating new data. The goals of such an adversary include impersonating a legitimate participant and learning information for a future attack. Second, we evaluate the privacy guarantees of our framework against honest but rational participants. Adversaries in this model participate honestly unless they are able to gain by deviating. Specifically, the participant gains something if he learns information of value about another participant. We start with some preliminary definitions and notation.

Table 5.1
PCL specification of roles for $\mathcal{Q}$, including the sets indicating prior knowledge (ϕ) and plaintext knowledge gained (θ) during run R of $\mathcal{Q}$.

$\mathbf{Serv_{Acc}} \equiv (RA, LA)[$ receive $\widehat{C}, \widehat{SP}, (\tau_r, \tau_l, o, read)$; $e_{sr} \leftarrow$ enc $\tau_r, pk(SP)$; send $\widehat{SP}, \widehat{RA}, e_{sr}$; receive $\widehat{RA}, \widehat{SP}, e_s$; $(i_r \parallel \mathsf{K}_r \parallel h')$:= dec $e_s, sk(SP)$; new z; send $\widehat{SP}, \widehat{C}, z$; receive $\widehat{C}, \widehat{SP}, h$; receive $\widehat{RA}, \widehat{SP}, e_{OT}$:= OT (i_r); $(\sigma_r \parallel \mathsf{K}_c \parallel valid)$:= dec e_{OT}, K_r; $e_{sl} \leftarrow$ enc $\tau_l, pk(SP)$; send $\widehat{SP}, \widehat{LA}, e_{sl}$; receive $\widehat{LA}, \widehat{SP}, e'_s$; $(i_l \parallel d \parallel exp \parallel T \parallel sig)$:= dec $e'_s, sk(SP)$; receive $\widehat{LA}, \widehat{SP}, (\lambda \parallel code_l[T])$:= PIR$(i_l, X(i_l))$; new m; $\iota_{lm} := \iota^{-\lambda m}$; send $\widehat{SP}, \widehat{RA}, \iota_{lm}$; receive $\widehat{RA}, \widehat{SP}, \iota_{lmg}$; v := eval $\sigma_r, \lambda, m, \iota_{lmg}, \widehat{p_{o.a.i}}, exp, sig$ $e_o \leftarrow$ enc $[o], \mathsf{K}_c$; send $\widehat{SP}, \widehat{C}, e_o$; $]SP()$	$\mathbf{Init_{Acc}} \equiv (SP, \tau_r, \tau_l, o, read, b, pwd_c)[$ send $\widehat{C}, \widehat{SP}, (\tau_r, \tau_l, o, read)$; receive $\widehat{SP}, \widehat{C}, z$; y := hash$(b \parallel pwd_c)$; h := hash$(z \parallel y)$; send $\widehat{C}, \widehat{SP}, h$; receive $\widehat{SP}, \widehat{C}, e_o$; $[o]$:= dec e_o, K_c $]C([o])$
	$\mathbf{Auth_{Role}} \equiv ()[$ receive $\widehat{SP}, \widehat{RA}, e_{sr}$; e_s := dec $e_{sr}, sk(RA)$; send $\widehat{RA}, \widehat{SP}, e_s$; send $\widehat{RA}, \widehat{SP}$, OT $(\cdot)$; receive $\widehat{SP}, \widehat{RA}, \iota_{lm}$; $\iota_{lmg} := (\iota_{lm})^{\gamma}$; send $\widehat{RA}, \widehat{SP}, \iota_{lmg}$; $]RA()$
	$\mathbf{Auth_{Loc}} \equiv ()[$ receive $\widehat{SP}, \widehat{LA}, e_{sl}$; e'_s := dec $e_{sl}, sk(LA)$; send $\widehat{LA}, \widehat{SP}, e'_s$; send $\widehat{LA}, \widehat{SP}$, PIR $(\cdot)$; $]LA()$

$\theta_C \overset{def}{=} \{\tau_r, \tau_l, \mathsf{K}_c, pwd_c, b, o, a\}$	$\phi_{C,R} \overset{def}{=} \{z, y, h, e_o, [o]\}$
$\theta_{SP} \overset{def}{=} \{\widehat{P_{o.a}}, \iota^{-1 \mod phi(N)}\}$	$\phi_{SP,R} \overset{def}{=} \{\tau_r, \tau_l, o, a, i_r, \mathsf{K}_r, h', z, h, \sigma_r, \mathsf{K}_c, valid, i_l, d, exp, T, \lambda, sig, m, \iota^{-\lambda m}, \iota^{-\lambda m \gamma}, \iota^{\rho x \lambda m}, \iota^{\rho \lambda m}, s, \iota^{\alpha\delta}, \iota^{\alpha\delta m} code_l[T], v, e_o\}$
$\theta_{RA} \overset{def}{=} \{\iota, \rho, \gamma, \phi(N), x, \sigma_r, \mathsf{K}_r, \mathsf{K}_c, valid, \tau_r, \}$	$\phi_{RA,R} \overset{def}{=} \{\iota^{-\lambda m}, \iota^{-\lambda m \gamma}\}$
$\theta_{LA} \overset{def}{=} \{\lambda, i_l, d, exp, sig, code_l[T], \tau_l\}$	$\phi_{LA,R} \overset{def}{=} \emptyset$

5.4.1 Definitions & Notation

A function f is **negligible** if, for any constant c, there exists a constant N such that $\forall\ n > N, f(n) < n^{-c}$. We write negl to indicate a negligible function. Given two values x and y, $x \overset{?}{=} y$ is conventional notation to refer to the question of the equivalence of x and y. We also write $\{x \overset{?}{=} y\}_P$ to indicate that a principal P is able to correctly answer the question. In our formalization and our proofs, we frequently refer to the **knowledge** that a principal has. For brevity, we omit IND-CPA-secure messages from this knowledge, as

$$\Pr[\{m \overset{?}{=} \widehat{m}\}_P \mid \mathsf{Has}(P, \{\mathsf{Enc}_K(m), \mathsf{Enc}_K(\widehat{m}), K, m\})] - \Pr[\{m \overset{?}{=} \widehat{m}\}_P \mid \mathsf{Has}(P, \emptyset)] \\ \leq \mathsf{negl}(n)$$

That is, the ciphertexts reveal no useful information to P, even if P knows the key used for both and one of the plaintexts. Similarly, we omit the details of messages exchanged in PIR and OT, as the privacy of those schemes is demonstrated in existing work.

Table 5.1 shows the translation of Protocol $\mathcal{Q}$ into protocol composition logic (PCL) behaviors[7] that indicate the actions taken by an honest participant. Due to space constraints, we will not provide an overview of PCL, and refer the reader to the work of Datta *et al.* [140]. Using this specification, we can model the knowledge gained by each participant in a formal manner. That is, we write $\mathsf{Has}(P, \theta_P)$ to indicate that P knows the value of all variables in the set θ_P. To describe the knowledge gained by P during run R of $\mathcal{Q}$, we use the proposition

$$\mathsf{Has}(P, \theta_P)\ [R]_P\ \mathsf{Has}(P, \phi_{P,R})$$

Our convention is to use θ for prior knowledge, while ϕ includes information gained during the execution. For simplicity, we assume $\theta_P \subseteq \phi_{P,R}$ (P does not forget the data in θ_P) and generally omit this notation for brevity. Table 5.1 formalizes these sets for honest parties in $\mathcal{Q}$. Note that, also for the sake of brevity, we omit public data and private keys from these sets. That is, public keys and the group $\mathbb{Z}_N^*$ are known to all parties at all times, while private keys are known only to the corresponding principal.

[7]The proper PCL terminology for what we call "behaviors" is "roles," which is problematic when discussing a protocol used for RBAC enforcement.

The first concern if our protocol is whether or not an eavesdropping adversary $\mathcal{A}$ can impersonate one of the principals by observing the execution by honest participants. To model intrusion, we start by defining the following impersonation experiment:

The impersonation experiment $\mathsf{Imp}^{P}_{\mathcal{A},\mathcal{Q}}$:

1. All parties execute run R of $\mathcal{Q}$, with $\mathcal{A}$ taking on the role of principal P.
2. Honest participants abort R if they have a suspicion of $\mathcal{A}$ has impersonated an honest principal.
3. All participants $\widehat{P}$ honestly reveal $\phi_{\widehat{P},R}$.
4. Upon request, each participant $\widehat{P}$ will decrypt any message e_K if $K \in \theta_{\widehat{P}}$.
5. The experiment outputs 1 if and only if no honest participant can demonstrate $\mathcal{A}$ has attempted to impersonate P. Otherwise, the output is 0.

We say **impersonation of P by $\mathcal{A}$ fails** if, for a sufficiently large positive integer n:

$$\Pr[\mathsf{Imp}^{P}_{\mathcal{A},\mathcal{Q}} = 1] \leq \frac{1}{n} + \mathsf{negl}(n)$$

That is, $\mathsf{Imp}^{P}_{\mathcal{A},\mathcal{Q}} = 1$ iff $\mathcal{A}$ can execute the behavior of P without detection by any other principals.[8] The parameter $1/n$ is used to account for $\mathcal{A}$ performing an extraordinary feat (*e.g.,* forging a cryptographic hash or encrypted message by blindly guessing). Additionally, the accusation of impersonation must be made *prior to* revealing $\phi_{\widehat{P},R}$.

Our other concern in this chapter is what information the protocol reveals to honest participants about C. To evaluate this goal, we define an additional experiment as follows:

[8]Clearly, we are not considering external factors such as IP addresses in the determination of $\mathsf{Imp}^{P}_{\mathcal{A},\mathcal{Q}}$, as our focus is on the protocol.

The privacy-preservation experiment $\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}}$:

1. All parties execute $\mathcal{Q}$, yielding the knowledge sets in Table 5.1. Let u denote the user's identity, r denote the role used, and l denote the user's location.
2. $\mathcal{A}$ guesses user $\widehat{u}$, role $\widehat{r}$, or location $\widehat{l}$.
3. The experiment outputs 1 if and only if $u = \widehat{u}$, $r = \widehat{r}$, or $l = \widehat{l}$. Otherwise, the output is 0.

We say a protocol **preserves the privacy of the client *C* against the adversary** $\mathcal{A}$ if,

$$\Pr[\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}} = 1 \mid \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R})] - \Pr[\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}} = 1 \mid \mathsf{Has}(\mathcal{A}, \emptyset)] \leq \mathsf{negl}(n)$$

That is, the adversary's chance of guessing the role, location, or identity of the user is negligibly different than blindly guessing.

5.4.2 Security Against Intruders

In this section, we consider security under the Dolev-Yao adversarial model. That is, $\mathcal{A}$ observes all data transmitted, can send and receives messages, store and retrieve data, and decrypt messages with known keys. We assume $\mathcal{A}$ begins with only public knowledge. As such, $\theta_{\mathcal{A}} = \emptyset$. Recall that our protocol is built on the assumption that underlying channels are encrypted, and we omitted this fact from the protocol definition for brevity and clarity. As such, $\mathcal{A}$ can only see the encrypted versions of data transmitted.

Lemma 1. Consider an eavesdropping adversary $\mathcal{A}$ observing run R of $\mathcal{Q}$. Given $\theta_{\mathcal{A}} = \emptyset$, the information gained by $\mathcal{A}$ (excluding OT and PIR) consists of the following pieces of data:

$$\phi_{\mathcal{A},R} = \{(\tau_r, \tau_l, o, a), e_{sr}, e_s, z, h, e_{sl}, e'_s, \iota^{-\lambda m}, \iota^{-\lambda m \gamma}, e_o\}$$

Proof. Follows from inspection. $\square$

Lemma 2. Consider adversary $\mathcal{A}$ attempting to execute run R' of $\mathcal{Q}$. Impersonation of RA by an eavesdropping adversary $\mathcal{A}$ fails.

Proof. By eavesdropping on R, the following proposition holds by Lemma 1:

$$\mathsf{Has}(\mathcal{A}, \emptyset)\ [R]_{\mathcal{A}}\ \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R})$$

In run R', $\mathcal{A}$ would receive $\widehat{e_{sr}}$. As $\mathcal{A}$ cannot determine if $\widehat{e_{sr}} \overset{?}{=} e_{sr}$, if $\mathbb{C}$ denotes the set of ciphertexts,

$$\Pr[\{\tau_r \overset{?}{=} \widehat{\tau_r}\}_{\mathcal{A}} \mid \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R} \cup \{\widehat{e_{sr}}\}) \leq \frac{1}{|\mathbb{C}|} + \mathsf{negl}(n)$$

That is, $\mathcal{A}$ cannot determine if the role tokens are the same. Consequently, $\mathcal{A}$ has two options. If $\mathcal{A}$ believes $\tau_r = \widehat{\tau_r}$, it can replay e_s; otherwise, it can generate a random guess $\widehat{e_s}$. Consider the former option. If $\tau_r = \widehat{\tau_r}$, when C generates $\mathsf{hash}(\widehat{b} \parallel pwd_c)$, it will only match the h' retrieved by *SP's* decryption of e_s if $\mathsf{hash}(\widehat{b} \parallel pwd_c) = \mathsf{hash}(b \parallel pwd_c)$, and, if $\mathbb{H}$ denotes the set of possible hash outputs,

$$\Pr[\neg\{\mathcal{A} \overset{?}{=} RA\}_C] \leq \frac{1}{|\mathbb{H}|} + \mathsf{negl}(n)$$

indicating that C will be able to identify the impersonation with all but trivial probability. Alternatively, if $\tau_r = \widehat{\tau_r}$, at the end of the protocol, $\mathcal{A}$ must compute $(\widehat{\iota^{-\lambda m}})^\gamma$ without knowledge of γ. Consequently, $\mathcal{A}$ must guess and

$$\begin{aligned}\Pr[\neg\{\mathcal{A} \overset{?}{=} RA\}_{SP}] &= \Pr[\mathsf{Has}(\mathcal{A}, \{\widehat{\iota^{-\lambda m\gamma}}\}) \mid \ \mathsf{Has}(\mathcal{A}, \{\widehat{\iota^{-\lambda m}}\}) \wedge \neg\mathsf{Has}(\mathcal{A}, \{\gamma\})] \\ &\leq \frac{1}{|Z_N^*|} + \mathsf{negl}(n)\end{aligned}$$

Thus, if $\mathcal{A}$ replays e_s, impersonation fails. Alternatively, if $\mathcal{A}$ believes $\tau_r = \widehat{\tau_r}$, it would generate a random $\widehat{e_s}$ and a random $\widehat{\iota^{-\lambda m\gamma}}$, which is unlikely as shown above. Thus, for a large n,

$$\Pr[\mathsf{Imp}^{RA}_{\mathcal{A},\mathcal{Q}} = 1] = \Pr[\neg\{\mathcal{A} \overset{?}{=} RA\}_C \wedge \neg\{\mathcal{A} \overset{?}{=} RA\}_{SP}] \leq \frac{1}{n} + \mathsf{negl}(n) \qquad \square$$

Lemma 3. Consider adversary $\mathcal{A}$ attempting to execute run R' of $\mathcal{Q}$. Impersonation of LA by an eavesdropping adversary $\mathcal{A}$ fails.

Proof. As in the proof of Lemma 2, $\mathcal{A}$ has the choice of replaying e'_s or generating a random $\widehat{e'_s}$. We have already seen that the latter course fails with near certainty. In the former option, similar to before,

$$\Pr[\{\tau_l \overset{?}{=} \widehat{\tau_l}\}_{\mathcal{A}} \mid \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R} \cup \{\widehat{e_{sl}}\}) \leq \frac{1}{|\mathbb{C}|} + \mathsf{negl}(n)$$

If $\tau_l = \widehat{\tau_l}$, then the signature required for verification will not match. That is,

$$\Pr[\mathsf{Imp}^{LA}_{\mathcal{A},\mathcal{Q}} = 1] = \Pr[sig \overset{?}{=} \widehat{sig}]$$
$$= \Pr[\mathsf{hash}(d \parallel code_l[T] \parallel \tau_r) = \mathsf{hash}(\widehat{d} \parallel \widehat{code_l[T]} \parallel \widehat{\tau_r})] \leq \frac{1}{|\mathbb{H}|} + \mathsf{negl}(n)$$

Thus, impersonation of LA by $\mathcal{A}$ fails. □

Lemma 4. Consider adversary $\mathcal{A}$ attempting to execute run R' of $\mathcal{Q}$. Impersonation of C by an eavesdropping adversary $\mathcal{A}$ fails.

Proof. As before, by eavesdropping on R, the following proposition holds by Lemma 1:

$$\mathsf{Has}(\mathcal{A}, \emptyset)\ [R]_{\mathcal{A}}\ \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R}) \supset \mathsf{Has}(\mathcal{A}, \{h\})$$

Note that $h = \mathsf{hash}(z \parallel y)$, where $z \in \phi_{\mathcal{A},R}$ and $y = \mathsf{hash}(b \parallel pwd_c) \notin \phi_{\mathcal{A},R}$. Furthermore, $pwd_c \notin \phi_{\mathcal{A},R}$. As such, given the assumption that the hash function is collision resistant, $\mathcal{A}$ must resort to guessing $\widehat{h}$, and

$$\Pr[\mathsf{Imp}_{\mathcal{A}}(C) = 1] = \Pr[\mathsf{Has}(\mathcal{A}, \{\widehat{h}\})] \leq \frac{1}{|\mathbb{H}|} + \mathsf{negl}(n)$$

Thus, impersonation of C by $\mathcal{A}$ fails. □

Lemma 5. Consider adversary $\mathcal{A}$ attempting to execute run R' of $\mathcal{Q}$. Impersonation of SP by an eavesdropping adversary $\mathcal{A}$ fails.

Proof. If $\mathcal{A}$ believes that the object $\widehat{o} = o$ and the action $\widehat{a} = a$, then the simplest strategy is to bypass Protocol $\mathcal{Q}$ and simply return e_o to C. Note, though, that $\mathcal{A}$ must send *something* to RA and LA under the terms of $\mathsf{Imp}^{SP}_{\mathcal{A},\mathcal{Q}}$ (otherwise, the impersonation would immediately be detected). It turns out that $\mathcal{A}$ can fool both RA and LA by sending random data, as neither principal ever sees a structured message in $\mathcal{Q}$. However, impersonation still fails, as, letting $n = |\mathcal{O}| \cdot |\mathcal{A}|$,

$$\Pr[\neg\{\mathcal{A} = SP\}_C] = \Pr[(o = \widehat{o}) \wedge (a = \widehat{a})] \leq \frac{1}{n} + \mathsf{negl}(n) \qquad \square$$

Lemma 6. Attempting to impersonate any principal yields no information useful for future attacks.

Proof. In the case of $\mathsf{Imp}_{\mathcal{A}}(RA)$ or $\mathsf{Imp}_{\mathcal{A}}(LA)$, note that $\phi_{\mathcal{A},R'} = \{\widehat{e_{sr}}, \widehat{\iota^{-\lambda m}}\}$ or $\{\widehat{e_{sl}}\}$, respectively. Without the corresponding secret key, $\mathcal{A}$ cannot decrypt the messages $\widehat{e_{sr}}$ and $\{\widehat{e_{sl}}\}$. Additionally, the IND-CPA security ensures that $\mathcal{A}$ cannot determine if this data decrypts identically as future such messages. Next, with no knowledge of $\iota, \iota^{-1}, \widehat{\lambda}$, or $\widehat{m}$, $\widehat{\iota^{-\lambda m}}$ is meaningless. Furthermore, $\widehat{m}$ is a nonce, so $\widehat{\iota^{-\lambda m}}$ provides no useful information. Now, consider $\mathsf{Imp}_{\mathcal{A}}(C)$. Observe that, in the middle of run R',

$$\mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R})\ [\mathsf{receive}\ \widehat{SP}, \widehat{C}, \widehat{z}]\ \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R} \cup \{\widehat{z}\})$$

Based on this information, $\mathcal{A}$ must be able to compute $\widehat{h}$, which we showed above was unlikely. Consequently, SP will detect the impersonation attempt when validating the hash and abort the protocol. Thus, with all but trivial probability,

$$\mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R})\ [R']\ \mathsf{Has}(\mathcal{A}, \phi_{\mathcal{A},R} \cup \{\widehat{z}\})$$

which contains no information useful for future attacks. If the hash guess is successful, the only additional information received by $\mathcal{A}$ is $\widehat{e_o}$. Again, this is based on an IND-CPA-secure encryption, and is not useful. Now consider $\mathsf{Imp}_{\mathcal{A}}(SP)$. We can formalize the maximum information learned by $\mathcal{A}$ as

$$\{(\tau_r, \tau_l, o, a), \widehat{e_{sr}}, \widehat{e_s}, \widehat{z}, \widehat{h}, \widehat{e_{OT}}, \widehat{e_{sl}}, \widehat{e'_s}, \widehat{\lambda}, \widehat{code_l[T]}, \widehat{r}\})$$

Again, most of the data (including τ_r and τ_l) is IND-CPA-secure or nonces, and provide no useful information for future attacks. The only pieces of data that could potentially be used for future attacks are $\widehat{\lambda}$ and $\widehat{code_l}[T]$. Since $i_l \notin \phi_{\mathcal{A},R'}$, $\widehat{\lambda}$ is simply a randomly selected location identifier $\lambda \in \mathbb{Z}_N^*$, which is public data. On the other hand, $\widehat{code_l}[T]$ is sensitive, as this could be used to forge the signature used in Protocol 2 by the *LD*. However, these are rolling codes and only valid for a short time frame, limiting the potential use by $\mathcal{A}$. Furthermore, forging the corresponding $\widehat{\tau_l}$ requires $i_l \in \phi_{\mathcal{A},R'}$, which is not the case. Consequently, this knowledge is useless for a future impersonation attack. □

Theorem 2. Protocol $\mathcal{Q}$ is secure against impersonation attacks under the Dolev-Yao adversarial model.

Proof. Follows directly from Lemmas 1-6. □

5.4.3 Preservation of Client Privacy

Before proceeding with the following lemmas, we emphasize here that our definition of privacy preservation focuses on the information exchanged in a single run of the protocol. As such, we do not consider attacks based on inference over time. That is, background knowledge of the unencoded policies may allow an adversary to aggregate the information gained from a significant number of access requests to break the privacy guarantees of our framework. At this time, we cannot determine whether there exists a computationally feasible approach that would mitigate this threat and leave such questions open for future consideration.

Lemma 7. $\mathcal{Q}$ preserves the privacy of C against RA.

Proof. Recall $\phi_{RA,R} = \{\iota^{-\lambda m}\}$ with λ and m as large values unknown to RA. Additionally, the mapping from λ to locations is unknown to RA. Thus, for a randomly selected $\widehat{l}$ and $\widehat{\lambda}$,

$$\begin{aligned}
&\Pr[\{l \stackrel{?}{=} \widehat{l}\}_{RA} \mid \mathsf{Has}(RA, \phi_{RA,R})] - \Pr[\{l \stackrel{?}{=} \widehat{l}\}_{RA} \mid \mathsf{Has}(RA, \emptyset)] \\
&\leq \Pr[\{\lambda \stackrel{?}{=} \widehat{\lambda}\}_{RA} \mid \mathsf{Has}(RA, \{\iota^{-\lambda m}\}) \wedge \neg\mathsf{Has}(RA, \{\lambda, m\})] \\
&\quad -\Pr[\{l \stackrel{?}{=} \widehat{l}\}_{RA} \mid \mathsf{Has}(RA, \emptyset)] \\
&\leq \mathsf{negl}(n)
\end{aligned}$$

Similarly, for role r and a random role $\widehat{r}$, as encryption is IND-CPA-secure,

$$\begin{aligned}
&\Pr[\{r \stackrel{?}{=} \widehat{r}\}_{RA} \mid \mathsf{Has}(RA, \phi_{RA,R})] - \Pr[\{r \stackrel{?}{=} \widehat{r}\}_{RA} \mid \mathsf{Has}(RA, \emptyset)] \\
&= \Pr[\{\tau_r \stackrel{?}{=} \widehat{\tau_r}\}_{RA} \mid \mathsf{Has}(RA, \{e_{sr}, \widehat{e_{sr}})] - \Pr[\{\tau_r \stackrel{?}{=} \widehat{\tau_r}\}_{RA} \mid \mathsf{Has}(RA, \emptyset)] \leq \mathsf{negl}(n)
\end{aligned}$$

As RA cannot determine the role session, it cannot determine the user either. Thus, $\mathcal{Q}$ preserves the privacy of C against RA. □

Lemma 8. $\mathcal{Q}$ preserves the privacy of C against LA.

Proof. Recall $\phi_{LA,R} = \emptyset$. Then, trivially,

$$\Pr[\mathsf{Priv}^{C}_{LA,\mathcal{Q}} = 1 \mid \mathsf{Has}(LA, \phi_{LA,R})] - \Pr[\mathsf{Priv}^{C}_{LA,\mathcal{Q}} = 1 \mid \mathsf{Has}(LA, \emptyset)] = 0 \leq \mathsf{negl}(n) \ \square$$

Lemma 9. $\mathcal{Q}$ preserves the privacy of C against colluding principals RA and LA.

Proof. Let $\mathcal{A}$ denote the adversary resulting from the collusion of RA and LA. Then,

$$\phi_{\mathcal{A},R} = \phi_{RA,R} \cup \phi_{LA,R} = \phi_{RA,R}$$

Observe that $\mathcal{A}$ now has (from prior and learned knowledge), ι, γ, and $\phi(N)$, but PIR ensures that $\mathcal{A}$ does not have λ. Furthermore, even with ι and $\phi(N)$, the intractability of the discrete logarithm prevents $\mathcal{A}$ from learning λm from $\iota^{-\lambda m}$. In addition, OT prevents $\mathcal{A}$ from learning the role token (and, as a result, the role and user) used. Consequently, pooling the knowledge of RA and LA is of no use, and

$$\Pr[\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}} = 1 \mid \mathsf{Has}(\mathcal{A}, \phi_{RA,R} \cup \phi_{LA,R})] - \Pr[\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}} = 1 \mid \mathsf{Has}(\mathcal{A}, \emptyset)] \leq \mathsf{negl}(n) \ \square$$

Lemma 10. Given the policy encoding scheme, *SP* can statically link two policies with the same role-location values with probability only negligibly better than guessing.

Proof. Observe that, as a result of the RSA assumption, *SP* cannot compute $((\rho\lambda)^{-1}(\alpha\delta))^{-1} \mod \phi(N)$, as the factorization of N is unknown to *SP*. This defense prevents linkability of policies across objects or actions. Consider the following three policies:

$$\widehat{p_{o.i}} = <(\rho\lambda)^{-1}(\alpha\delta), \iota^{\alpha\delta}> \widehat{p_{o.j}} = <(\rho\lambda)^{-1}(\alpha\widehat{\delta}), \iota^{\alpha\widehat{\delta}}>$$
$$\widehat{p_{o.k}} = <(\widehat{\rho\lambda})^{-1}(\alpha\widehat{\delta}), \iota^{\alpha\widehat{\delta}}>$$

Without knowledge of $\phi(N)$, *SP* cannot compute the multiplicative inverses and link $\widehat{p_{o.i}}$ with $\widehat{p_{o.j}}$. Furthermore, without knowledge of any of the factors or ι, *SP* cannot distinguish between $\widehat{p_{o.j}}$ and $\widehat{p_{o.k}}$ given knowledge of $\widehat{p_{o.i}}$. Thus,

$\Pr[\{p_{o.i}[r] \overset{?}{=} p_{o.j}[r]\}_{SP} \vee \{p_{o.i}[l] \overset{?}{=} p_{o.j}[l]\}_{SP} \mid \mathsf{Has}(SP, \{\widehat{p_{o.i}}, \widehat{p_{o.j}}\})] -$
$\Pr[\{p_{o.i}[r] \overset{?}{=} p_{o.j}[r]\}_{SP} \vee \{p_{o.i}[l] \overset{?}{=} p_{o.j}[l]\}_{SP} \mid \mathsf{Has}(SP, \emptyset)] \leq \mathsf{negl}(n)$ □

Lemma 11. $\mathcal{Q}$ preserves the privacy of *C* against *SP* for a single access request.

Proof. As $\lambda \in \phi_{SP,R}$, *SP* learns a small amount of information about *C*. However, without the mapping from λ to locations (known only to *LA*), for a randomly selected $\widehat{l}$,

$$\Pr[\{l \overset{?}{=} \widehat{l}\}_{SP} \mid \mathsf{Has}(SP, \phi_{SP,R})] - \Pr[\{l \overset{?}{=} \widehat{l}\}_{SP} \mid \mathsf{Has}(SP, \emptyset)] \leq \mathsf{negl}(n)$$

Similarly, *SP* does not have the mapping from ρ to roles. Furthermore, $\rho \notin \phi_{SP,R}$. Hence,

$$\Pr[\{r \overset{?}{=} \widehat{r}\}_{SP} \mid \mathsf{Has}(SP, \phi_{SP,R})] - \Pr[\{r \overset{?}{=} \widehat{r}\}_{SP} \mid \mathsf{Has}(SP, \emptyset)] \leq \mathsf{negl}(n)$$

Finally, consider an encoded policy

$$\widehat{p_{o.i}} =< (\rho\lambda)^{-1}(\alpha\delta), \iota^{\alpha\delta} >$$

As $\rho, \alpha, \delta, \iota, \phi(N) \notin \phi_{SP,R}$, *SP* cannot determine if this was the policy satisfied, even if it knows (by organizing the storage of policies) that $\iota^{\alpha\delta}$ indicates the object-action pair. Consequently, *SP* cannot trace $\phi_{SP,R}$ back to the original, unencoded $p_{o.i}$ that was satisfied. Thus,

$$\Pr[\mathsf{Priv}^{C}_{SP,\mathcal{Q}} = 1 \mid \mathsf{Has}(SP, \phi_{SP,R})] - \Pr[\mathsf{Priv}^{C}_{SP,\mathcal{Q}} = 1 \mid \mathsf{Has}(SP, \emptyset)] \leq \mathsf{negl}(n) \qquad \square$$

Lemma 12. Given runs R and R' of $\mathcal{Q}$ made with location tokens τ_l and $\widehat{\tau_l}$, *SP* can link the requests if the location is identical.

Proof. *SP* always sees λ in the clear, so the claim trivially holds when λ is identical. $\square$

Corollary 12.1. Location linkage threats can be mitigated by administrative action.

Proof. At regular intervals, *IA* can regenerate the λ values as λ'. *IA* can provide *LA* with an updated mapping from λ' to location. Simultaneously, *IA* can regenerate the encoded policies $\widehat{p_{o.i}}$, sending the updated versions to each *SP*. While this process requires significant work on the part of *IA*, the policy regeneration can be performed offline before the update takes effect. Thus, *IA* can mitigate the threat of location linkage over time. $\square$

Lemma 13. Given runs R and R' such that $\rho = \widehat{\rho}$ and $\lambda = \widehat{\lambda}$, where ρ, λ are used in R and $\widehat{\rho}, \widehat{\lambda}$ are used in R', there is a strategy by which *SP* can link these requests, but it is probabilistically detectable by *RA*.

Proof. Assume $\lambda = \widehat{\lambda}$. If *SP* repeats the value m, then $\phi_{SP,R} \cup \phi_{SP,R'}$ contains $\iota^{\rho\lambda m}$ and $\iota^{\widehat{\rho}\lambda m}$. Consequently, it is trivial for *SP* to determine if $\rho = \widehat{\rho}$. Thus, by repeating m when λ is the same, *SP* can state with absolute certainty that both requests used the same role and location. However, $\iota^{-\lambda m} \in \phi_{RA,R} \cap \phi_{RA,R'}$. Consequently, *RA* can state with absolute certainty that $-\lambda m = -\widehat{\lambda}\widehat{m}$, though this does not necessarily indicate the m values are identically. However, if c denotes the cardinality of the congruence class $[\widehat{m}]_{\bmod \phi(N)}$,

$$\Pr[-\lambda m = -\widehat{\lambda}\widehat{m} \mid \text{ only } \widehat{m} \text{ is random}] = \frac{c}{|\mathbb{Z}_N^*|}$$

As this probability is small, such an equivalence is a likely indication of a repeated m, providing *RA* with a strong probabilistic guess that *SP* is not acting honestly.

Corollary 13.1. Collusion by multiple *SPs* is detectable by *RA*.

Proof. Follows from Lemma 13, under the premise that R and R' (and additional such runs) are runs by the distinct, colluding *SPs*. Other than λ (which is not useful without the ability to map it to a location or a policy), the runs reveal no useful data unless m is identical. As such, *RA* can detect the repeated m value. □

Lemma 14. Assuming *RA* prevents reuse of m for the same locations, with the exception of location, $\mathcal{Q}$ preserves the privacy of *C* against *SP*, even over time.

Proof. First, note that repeated runs of $\mathcal{Q}$ from the same location l will repeatedly reveal λ to *SP*. Consider the sequence of runs $R_1, \ldots, R_n$ that satisfy policies $p_1, \ldots, p_n$, where λ is identical in all runs. If there is only a single location that satisfies $p_1, \ldots, p_n$, then *SP* can determine this location. However, from Lemmas 11-13 and Corollary 13.1, *SP* can only link the roles used in requests by repeating m for the same location value λ. Also, *RA* can detect this attempt with near certain probability (and a negligible false positive rate). As such, if *RA* prevents reuse (either by blacklisting corrupt *SPs* or by rejecting and requesting a new m), then *SP* can only determine that the location used in multiple requests were the same. Consequently, if we modify the privacy-preserving experiment to eliminate the discussion of location,

$$\Pr[\mathsf{Priv}^{C}_{SP,\mathcal{Q}} = 1 \mid \mathsf{Has}(SP, \bigcup \phi_{SP,R_i})] - \Pr[\mathsf{Priv}^{C}_{SP,\mathcal{Q}} = 1 \mid \mathsf{Has}(SP, \emptyset)] \leq \mathsf{negl}(n) \ \square$$

Lemma 15. $\mathcal{Q}$ fails to preserve the privacy of C against an adversary $\mathcal{A}$ that consists of SP who colludes with either RA or LA.

Proof. If SP colludes with RA, SP could simply send τ_r to RA, who decrypts the token to determine i_r, the index of the role session. RA can then retrieve σ_r, and use this information to find the corresponding ι^ρ, which RA can use to uniquely identify the role. Thus, $\mathcal{A}$ is guaranteed to correctly state $r = \widehat{r}$, and

$$\Pr[\mathsf{Priv}^{C}_{\mathcal{A},\mathcal{Q}} = 1] = 1$$

If SP colludes with LA, SP can send τ_l to LA with similar results. Therefore, $\mathcal{Q}$ fails to preserve the privacy of C against an adversary $\mathcal{A}$.

Theorem 3. Assuming SP does not collude with RA or LA, $\mathcal{Q}$ preserves the privacy of C against SP, RA, and LA.

Proof. The claim follows from above. $\square$

5.5 Summary of Protocol Properties

In the setting of formal proofs, it is easy to lose sight of the big picture of the proposed scheme. While the previous section presents a number of formal proofs for properties of our framework, we can summarize these in more natural language as follows.

- RA fails to learn which user, role, or location is involved in the request. For multiple distinct requests, RA is prevented from determining if the user, role, or location is the same in both.
- Similarly, LA fails to learn which user, role, or location is involved in the request. For multiple distinct requests, LA is prevented from determining if the user, role, or location is the same in both.

- Encrypted storage of role session records ensures that *SP* can only retrieve information for which it has an authentic τ_r.
- The policy encoding scheme ensures that *SP* cannot perform a static analysis to link encoded policies that have common characteristics (*e.g.,* same role, location).
- Protocol $\mathcal{Q}$ protects the user's location information in the short term. Over time, *SP* may be able to determine the location by linking policies. However, without additional information, *SP* cannot link this to a particular user. Furthermore, *IA* can mitigate this threat by updating the λ values at regular intervals.
- Unless *SP* colludes with *RA* or *LA*, $\mathcal{Q}$ protects the privacy of the user's identity and role use, *even over time.*

5.6 Conclusions

In this chapter, we proposed a privacy-preserving framework for evaluating spatially aware RBAC policies. We defined an architecture with reasonable computation assumptions, and specified protocols for evaluating the requests. We formally modeled the main request protocol using PCL. Using this formal model, we proved that our framework is secure against external attacks, and we also proved that our protocols preserve the privacy of users for individual requests. In addition, assuming regular system maintenance is applied, the protocols can even protect users' privacy over time. Finally, we highlighted the similarities and differences between our scheme and attribute-based encryption, with the main difference being that our scheme requires *transient* credentials that are possessed by users for a single request at a time before reuse by others. In contrast, attribute-based encryption uses static credentials that persist for a single user.

6 ENFORCING PHYSICALLY RESTRICTED ACCESS CONTROL FOR REMOTE FILES

In the previous chapters, we focused on the issue of enforcing spatial constraints in an RBAC setting. Although our work answered a number of interesting questions, there was an extensive body of works that defined the formal underpinnings of these systems. In this chapter and the next, we embark on an exploration into a new field that lacks such a theoretical framework. Specifically, we consider the enforcement of access control policies, where the contextual factor is the device being used, rather than the user's location. To accomplish this goal, we integrate the physical properties of a device directly into the access control framework. Our exploration involves adapting cryptographic protocols to use the unique properties of physically unclonable functions (PUFs).

6.1 Motivation for Physically Restricted Access Control

Controlled remote access to protected resources is a critical element in security for distributed computing systems. Often, some resources are considered more sensitive than others, and require greater levels of protection. Recent advances in access control [17,18,34] provide means to tighten the security controls by considering users' contextual factors. While these techniques offer more fine-grained control than traditional identity-based approaches, we desire an even stronger guarantee: Our goal is to provide a means by which access is granted only to known, trusted devices.

To achieve our aim, we had to address two separate issues. First, we required the ability to identify a device uniquely. That is, our scheme must be able to distinguish between two devices with software that is configured identically. Second, we had to establish a mechanism for encrypting the data for access by only the identified device.

A naïve approach to this problem would be to apply authentication mechanisms at the network and transport layers, for instance Challenge-Handshake Authentication Protocol (CHAP), Transport Layer Security (TLS), or Internet Protocol Security (IPsec). However, these solutions fail to provide our desired security guarantees in three ways. First, they differentiate based on stored data, *e.g.,* cryptographic keys. If this data is leaked, these solutions can be broken. Our approach, however, does not rely on the security of data stored on the client. Second, these approaches are too coarse-grained, granting or denying access below the application layer. That is, our solution allows a server program to selectively grant access to subsets of data based on the unique hardware of the remote device. Existing approaches cannot provide this flexibility.

A final shortcoming of these basic approaches is that they can be completely bypassed by improper management and insider threats, which remain a real and underestimated problem [141–146] Interestingly, in a recent report [147], the most common cause (48%) of data breaches was privilege misuse, which includes improper network configuration and malicious insider threats. In our approach, access control decisions are based on the physical properties of the remote devices themselves, and are not dependent upon network configuration settings. While this does not completely eliminate insider threats, our solution does offer a higher level of defense against such insider threats.

Alternatively, one could rely on a public key infrastructure (PKI) using trusted platform modules (TPMs). While these approaches will work in traditional computing environments, our interests extend to environments for which TPMs are not available or PKI is considered to be too expensive. Specifically, we desire a solution that could also be deployed in low-power embedded systems. In these scenarios, the computing power required for modular exponentiation can quickly exhaust the device's resources. Furthermore, PKI schemes that are based on stored keys can be attacked by extracting the key; a presentation at BlackHat 2010 demonstrated how to do so [148]. Our approach relies on a cryptographic scheme that offers similar guarantees as PKI, but with less computation required. In addition, our solution does not use persistently stored keys, thus reducing the attack surface.

Our solution [149, 150] is based on the use of physically unclonable functions (PUFs) [52, 53]. PUFs rely on the fact that it is physically impossible to manufacture two identical devices. For example, two application-specific integrated circuits (ASICs) can be manufactured on the same silicon wafer, using the same design. However, a circuit in one ASIC may execute faster than the equivalent circuit in the other, because the wire length in the first is a nanometer shorter than the second. Such variations are too small to control and can only be observed during execution. PUFs quantify these variations as challenge-response pairs, denoted (C, R), that are unique to each particular hardware instance. A robust PUF is unpredictable, yet consistent for a single device. It is also unforgeable, as the physical variations that determine the PUF are too small to control.

Previous works on PUFs have focused on two areas. First, PUFs can be used to store cryptographic keys in a secure manner. Given a PUF pairing (C,R) and a key K, the device stores $X = R \oplus K$. In this case, R acts as a one-time pad, and X is a meaningless string of bits that can be stored in plaintext on a hard-drive. When the key is needed at a later time, the device again executes the PUF to get R and recovers the key as $K = R \oplus X$. The second use of PUFs is to generate cryptographic keys directly by mapping R to, for example, a point on an elliptic curve. In such a usage, the PUF does not have to store any data.

The advantages of employing PUFs for key generation and storage are subtle, and may be missed at first glance. First, note that no cryptographic keys are explicitly stored; the only data above that is ever stored is the value X, which is a random, meaningless bit string that reveals no information regarding the key K. A second advantage, which follows from the first, is that any key exists only at run-time. Furthermore, if the PUF is integrated into the processor itself, then the keys never even exist in main memory. Thus, PUFs offer very strong protections of cryptographic keys.

While these previous works assume a traditional cryptographic scheme is in place, we propose a new and unique direction for PUF research. That is, we propose incorporating the randomness of the PUF directly into an application-layer access request protocol. Our light-weight multifactor authentication mechanism, coupled with a dynamic key generation scheme, provides a novel technique for enforcing access control restrictions based on the device used.

6.2 Threat Model

In describing our threat model, we start with the central server S. We first note that the adversary's goal is to gain access to sensitive data stored on S. We place no restrictions on what constitutes this data; we simply note that a server application running on S is responsible for the access control decisions. Next, we assume that S is trusted and secure. While this may seem like a strong assumption to make, we stress that it is the *data* stored on S that is important. That is, if an adversary can compromise S, there is no need to attack our protocols, as he has already "won."

Regarding the client devices C, we assume that the organization has the authority to tightly control the software running on each device. While this is a daunting task for traditional computing, recall that we are also highly motivated by the concerns of embedded distributed applications. Embedded devices do not require the complex code base that exists in a traditional workstation; thus, satisfying this requirement is easier. Furthermore, our protocols will still apply in traditional schemes, too. Specifically, remote attestation techniques can be used to ensure that only known, trusted software is running.

Our main adversaries, then, are the users. We consider two classes of users as threats. First, client users have full access to the device, with the exception of installing software. That is, these users can read any data stored on the device. However, they cannot extract the data from memory to external storage. Also note, in the case of embedded systems, there might not actually be a user, as the devices may be executing autonomously. If there is a human user, he will have a password, and we assume it is protected.

The other class of users that pose a threat, whether malicious or not, are administrators. While administrators may have access to the data on S directly, our assumption is that the goal of a malicious administrator is to enable access to an untrusted device, thereby bypassing the physical restrictions. This adversary has access to all secret data stored on S.

Finally, we also consider network-based attackers, such as eavesdroppers. In all cases, we apply standard cryptographic assumptions. Specifically, we assume that adversaries are limited to probabilistic, polynomial-time attacks.

6.3 PUFs

The fundamental idea of PUFs is to create a random pairing between a challenge input C and a response R. The random behavior is based on the premise that no two instances of a hardware design can be identical. That is, one can create a PUF by designing a piece of hardware such that the design is intentionally non-deterministic. The physical properties of the actual hardware instance resolve the non-determinism when it is manufactured. For example, the length of a wire in one device may be a couple of nanometers longer than the corresponding wire in another device; such differences are too small to be controlled and arise as natural by-products of the physical world.

While there are several types of PUFs, in this work we focus on PUFs derived from ring oscillators (ROs). Figure 6.1 shows a sample 1-bit RO PUF. A RO consists of a circular circuit containing an odd number of not-gates; this produces a circuit that oscillates between producing a 1 and 0 as output. In a 1-bit PUF, the output of two ROs pass through a pair of multiplexors (MUX) into a pair of counters that count the number of fluctuations between the 0 and 1 output. The PUF result is 1 if

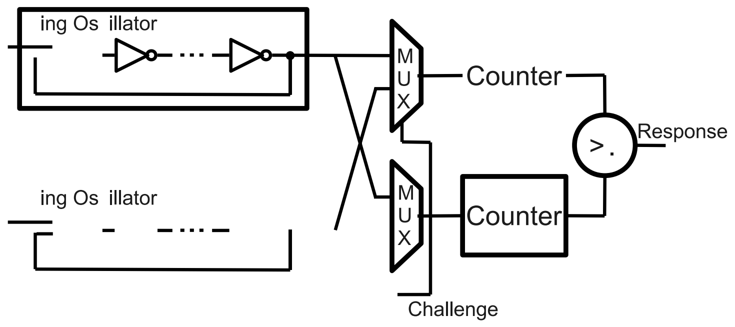


Figure 6.1. A sample 1-bit PUF based on ring oscillators

the counter on top holds a greater value, and 0 otherwise. The role of the challenge in a 1-bit RO PUF is to flip the MUX.

Clearly, it is not desirable to have such a one-to-one corresponence for larger PUFs. As such, for larger output bit strings it is better to have a pool of ROs, and randomly select pairs for comparison based on the challenge. In [54], the authors evaluate the entropy resulting from random pairings of ROs, and show that N ROs can be used to produce $\log_2(N!)$ bits. For example, 35 ROs can be used to create 133 bits. Thus, a small number of ROs can be used to exhibit good random behavior. Another way to introduce entropy into the PUF behavior is to apply a cryptographic hash to the output. Given a strong hash function, changing a single bit of the PUF challenge, which yields a single flipped PUF bit, will produce a very different output.

The interesting properties of PUFs arise from the fact that it is virtually impossible for two ROs to operate at the same frequency. Specifically, miniscule variations in the wire width or length can cause one RO to oscillate at a faster speed than the other. As these variations are persistent, one of the oscillators will *consistently* be faster. Thus, the behavior of PUFs based on ROs depends on the physical instance of the device. Also, if the PUF is large enough, the behavior is unique. Furthermore, as these variations can be neither predicted nor controlled, they cannot be cloned.

With the exception of our implementation description in Section 7.4, we will assume an *idealized PUF* in our protocol design. That is, given a challenge-response pair $< C_i, R_i >$ and another challenge $C_j = C_i$, one cannot predict the value of R_j. Consequently, our results apply to any PUF that meets this ideal, rather than just RO-based PUFs.

6.4 Physically Restricted Access Control

In this section, we start by defining our notion of physically restricted access control. Next, we offer a high-level protocol and formal analysis for achieving this goal. We then present a more concrete example of this protocol that is derived from the Feige-Fiat-Shamir identification scheme.

We assume that the protected resources consist of files on a central server and subjects request access to these files remotely. For a file access request by a subject from a given device, the access control system checks whether the subject is allowed to access the file from the device; if this is the case, the server encrypts the file with a dynamically generated key and sends the resulting data to the device.

We thus assume an access control model based on a number of sets. Let $\mathcal{S}$ denote the set of subjects, $\mathcal{D}$ the set of trusted devices, $\mathcal{F}$ the set of protected files, and $\mathcal{R}$ the set of privileges. For simplicity, we assume $\mathcal{R} = \{read, write\}$. A permission can be written as the tuple $< s, f, r >$, such that $s \in \mathcal{S}, f \in \mathcal{F}$, and $r \in \mathcal{R}$. Thus $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{F} \times \mathcal{R}$ defines the set of authorized permissions subject to the physical restrictions. Let $PUF_d : C \rightarrow R$ be the PUF for a trusted device $d \in \mathcal{D}$.

We define **physically restricted access control** to be the restriction of an access request $< s, d, f, r >$, subject to the following conditions:

- The identity of s is authenticated.
- $< s, f, r > \in \mathcal{P}$.
- $d \in \mathcal{D}$, and the authentication is performed implicitly by the ability of d to demonstrate a one-time proof of knowledge of PUF_d.

- A dynamic encryption key k_{PUF} based on the proof of PUF_d is used to bind the request to the device.

An important element of this definition is the notion of *hardware binding* of the cryptographic key. That is, the key k_{PUF} is generated dynamically and relies on the physical properties of the hardware itself (*i.e.,* the PUF). Consequently, k_{PUF} is never explicitly stored on the requesting device. This dynamic key generation is in contrast to traditional key management, in which keys are generated *a priori.* This approach simplifies the administration work, while reducing the threat of a rogue administrator transferring keys to an untrusted device.

One possible criticism to our definition is that it does not consider what happens to the contents of the file after decryption. That is, if the device d is malicious (or is infected with malicious software), it could simply broadcast the contents after decryption. We counter this objection by noting that remote attestation techniques could be applied to ensure that only trusted applications are running on the device. Hence, we assume either the device is free of malware, or the server can detect the malware and abort.

In addition to such software attacks, an attacker with physical access and sufficient technical skill could read the contents directly from the device's memory. However, such an attack exists regardless of the access control methodology applied. As such, we consider such threats beyond the scope of our work.

6.4.1 Protocols

Our protocols rely on a number of cryptographic primitives. Let H denote a collision-resistant hash function, while $\mathsf{Enc}_k(m)$ denotes the symmetric key encryption of a message m with the key k, using a cipher that is secure against *probabilistic polynomial time* (PPT) known ciphertext attacks. Define $\mathsf{Auth}(\cdot)$ to be a robust authentication scheme that is resilient against PPT adversaries. $\mathsf{Gen}(\cdot)$ denotes a pseudorandom key generator based on the provided seed value.

Table 6.1
Protocols for enforcing physically restricted access control

$\mathsf{Request}(adm, m)$ – Administrator adm requests m challenges to enable a new device.
– S performs $\mathsf{Auth}(adm)$
– S responds with $C_1, \ldots, C_m$, parameters $prms$, and a nonce n
$\mathsf{Enroll}(adm, pwd, C_1, \ldots, C_m, prms)$ – C (after receiving data provided by adm) sends a commitment of the PUF to S.
– S performs $\mathsf{Auth}(adm)$
– C generates $otk \leftarrow \mathsf{Gen}(pwd, n, C_1, \ldots, C_m)$
– C provides $\mathsf{Enc}_{otk}(\mathsf{Commit}(< C_1, R_1 >, \ldots, < C_m, R_m >))$
– S responds with $\mathsf{H}(\mathsf{Commit}(< C_1, R_1 >, \ldots, < C_m, R_m >))$
$\mathsf{Access}(user, file, action)$ – Subject $user$ requests $action$ for $file$, which is encrypted with key $chal$ and transferred. If $action = read$, S sends the file. Otherwise, C sends it.
– S performs $\mathsf{Auth}(user)$ and issues Chal (T), where $T \subset \mathscr{P}(C_1, \ldots, C_m)$
– S responds with a nonce z
– S verifies that $user$ is permitted to perform $action$ on $file$
– Generate and transfer $\mathsf{Enc}_{chal}(c)$, where $p \leftarrow \mathsf{Prove}(T)$ and $chal \leftarrow \mathsf{Gen}(p, z, pwd)$

Let $\mathsf{Commit}(\cdot)$ denote a commitment scheme that ensures confidentiality against PPT adversaries. $\mathsf{Chal}(\cdot)$ and $\mathsf{Prove}(\cdot)$, then, indicate a random challenge and the corresponding zero-knowledge proof of the secret value bound to the commitment. Furthermore, we assume that any PPT adversary $\mathcal{A}$ has negligible probability of guessing $\mathsf{Prove}(\cdot)$ without access to the committed secret value. Assuming C denotes the PUF-enabled client (also called the device) and S indicates the server, the table in Table 6.1 gives the formal definition of our protocols.

Given these formalisms, we now explain the intuition behind each protocol. In $\mathsf{Request}(adm, m)$, an administrator adm requests a set of m challenges to be used with a new (unspecified) device.[1] S authenticates adm and creates a database entry of the form $< adm, n, C_1, \ldots, C_m >$, binding those challenges and the nonce to that administrator. Hence, only that administrator is authorized to use that particular set

[1]In general, we assume $C_i \leftarrow \mathsf{Gen}(1^n)$ $\forall$ $1 \leq i \leq m$; that is, each challenge is the result of a pseudorandom generator with a security parameter 1^n. However, in some applications, it may be desirable for S to define the challenges predictably. As such, we are intentionally vague on the selection of $C_1, \ldots, C_m$.

of challenges. We use $prms$ to indicate any parameters needed for the commitment and proof scheme. For instance, in our implementation $prms$ consists of a modulus.

For $\mathsf{Enroll}(adm, pwd, C_1, \ldots, C_m, prms)$, we are assuming a trusted path from adm to C. That is, no eavesdropper learns the administrator's password, and all data are entered correctly. Based on this assumption, adm provides the inputs to C, which initiates an enrollment protocol that starts with authenticating adm. C uses a pseudorandom generator to produce a one-time-use key otk derived from the administrator's password pwd, the nonce n, and the challenges. S can retrieve the nonce and challenges from its database, thus recreating the key on its end. C uses otk to encrypt a commitment of the PUF challenge-response pairs. S acknowledges receipt of the values with a hash of the commitment.

Finally, $\mathsf{Access}(user, file, action)$ defines the access request protocol. As before, S authenticates the user making the request, and selects a random set T of the challenges $C_1, \ldots, C_m$. After receiving $\mathsf{Chal}(T)$, C executes the PUF to get the responses R_i for each $C_i \in T$. The corresponding zero-knowledge proof $p \leftarrow \mathsf{Prove}(T)$ is derived from these responses. S uses p and the user's password pwd as inputs to a pseudorandom generator to produce a one-time-use key k. S encrypts the file contents c with this key, returning the encrypted file to C. Hence, the intuition behind this protocol is that the file can only be decrypted by that user with that particular PUF-enabled device.

We note that there is one important consideration regarding our definition of $\mathsf{Access}(user, file, action)$. Unlike the previous protocols, this protocol will be executed repeatedly. However, there are only 2^m subsets of $\mathscr{P}(C_1, \ldots, C_m)$. After all subsets are exhausted for a single user, the necessary proof will be reused. However, this repetition is acceptable, as the proof is never made public. Instead, the proof is used as an input to the key generation. Furthermore, assuming the nonce z is never repeated, the keys generated will always be different, even if $p \leftarrow \mathsf{Prove}(T)$ is reused.

In designing our protocols, we envisioned both traditional computing and embedded applications. In the embedded scenario, there may not be a human $user$ making the request $\mathsf{Access}(user, \mathit{file}, action)$. A straightforward variant of our protocol could accommodate this situation by eliminating $\mathsf{Auth}(user)$ from that protocol. Then, S must make the access control decision based on the device making the request, not the *user* doing so. Though this flexibility is a nice feature of our design, we will not investigate the security claims of this variant in this paper.

6.4.2 Security Analysis

Here, we present our formal analysis of the security properties of our protocols. We start with three lemmas, and complete our analysis with a theorem that our approach satisfies our definition of physically restricted access control.

Lemma 1. A PPT adversary $\mathcal{A}$ can enable an untrusted device with only negligible probability.

Proof. Based on our assumption that $\mathsf{Auth}(\cdot)$ is resilient against PPT adversaries, S will abort the $\mathsf{Request}(\cdot)$ and $\mathsf{Enroll}(\cdot)$ protocols, except with negligible probability. Even with a transcript of $\mathsf{Request}(\cdot)$, $\mathcal{A}$ must be able to forge the $\mathsf{Enc}_{otk}(\cdot)$ message to enable an untrusted device. However, with no knowledge of pwd, this feat is also infeasible, by our assumptions of $\mathsf{Enc}_k(\cdot)$. Therefore, $\mathcal{A}$ has only negligible probability of completing the $\mathsf{Enroll}(\cdot)$ protocol and enabling an untrusted device. $\square$

Lemma 2. An honest client C can validate its enrollment with the legitimate S, except with negligible probability.

Proof. Similar to Lemma 1, a PPT adversary $\mathcal{A}$ has negligible probability of forging $\mathsf{H}(\mathsf{Commit}(< C_1, R_1 >, \ldots, < C_m, R_m >))$. Hence, if C receives such a hash, it has high assurance that the hash originated from the legitimate S and the enrollment succeeded. $\square$

Lemma 3. A PPT adversary $\mathcal{A}$ with transcripts of $\mathsf{Request}(\cdot)$ and $\mathsf{Enroll}(\cdot)$ can model the PUF with only negligible probability.

Proof. In order for $\mathcal{A}$ to learn the commitments of the PUF behavior, $\mathcal{A}$ must either decrypt $\mathsf{Enc}_{otk}(\mathsf{Commit}(< C_1, R_1 >, \ldots, < C_m, R_m >))$ or find a preimage of $\mathsf{H}(\mathsf{Commit}(< C_1, R_1 >, \ldots, < C_m, R_m >))$. However, based on our assumptions regarding $\mathsf{Enc}_k(\cdot)$ and $\mathsf{H}(\cdot)$, both actions are infeasible. Thus, these protocols do not leak enough information for a PPT adversary $\mathcal{A}$ to model the PUF. □

Informally, these lemmas demonstrate that the $\mathsf{Request}(\cdot)$ and $\mathsf{Enroll}(\cdot)$ protocols guarantee integrity and confidentiality against PPT adversaries. That is, by viewing a transcript of both protocols, $\mathcal{A}$ fails to learn the administrator's *pwd* or the PUF challenge-response pairs. Furthermore, any tampering by $\mathcal{A}$ will be detected by either S or C. Also, $\mathcal{A}$ cannot launch a man-in-the-middle attack against $\mathsf{Enroll}(\cdot)$, as doing so requires knowledge *pwd*. Applying these lemmas, we propose the following theorem.

Theorem 1. The $\mathsf{Access}(\cdot)$ protocol enforces physically restricted access control under the PPT adversarial model.

Proof. By Lemma 1, we are guaranteed that only trusted devices will be able to produce $p \leftarrow \mathsf{Prove}(T)$. Lemma 2 ensures that trusted devices receive confirmation if their enrollment is successful; as such, if the confirmation is not received, proper mitigation can be performed. By Lemma 3, we are guaranteed that PPT adversaries cannot possess a model of the PUF behavior by observing a transcript of the $\mathsf{Request}(\cdot)$ and $\mathsf{Enroll}(\cdot)$ protocols. We explicitly model the authentication of *user*, check that *user* is authorized to perform *action* on $file$, and the device is implicitly authenticated by generating a one-time proof of knowledge of the PUF behavior. Furthermore, the one-time key $chal \leftarrow \mathsf{Gen}(p, z, pwd)$ exists only at run-time, is never transmitted, is bound to the hardware of the requesting (trusted) device (by the use of the PUF), and is used to encrypt data transferred between C and S. The probability of a PPT adversary generating *chal* is negligible, so the encryption successfully enforces the access control policy. Therefore, by definition, the $\mathsf{Access}(\cdot)$ protocol enforces physically restricted access control under the PPT adversarial model. $\square$

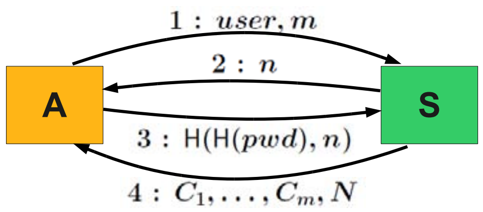
6.5 Implementation

In this section, we describe our implementation of a PUF-based access control mechanism based on our protocols described above. We start by describing our protocol instantiation and our implementation of a PUF using ring oscillators, which is the same method used in [54]. We also describe the use of Reed-Solomon codes to ensure the PUF produces a consistent result that can be used for authentication, and detail our minimal storage requirements.

6.5.1 Protocol Instantiation

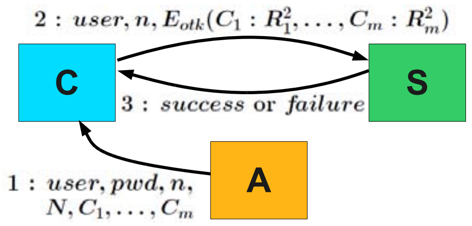
The underlying premise of our protocol instantiation is the Feige-Fiat-Shamir identification scheme. Our choice of hash function was SHA-1, although a better choice would be SHA-256, which offers more protection against preimage attacks and is collision-resistant. Our choice of symmetric key cryptography was AES which also provides the security against PPT adversaries that we require.

Our $\mathsf{Auth}(\cdot)$ primitive uses the hash function and a nonce n in a challenge-response protocol. Specifically, S generates n, and the user must respond with $\mathsf{H}(\mathsf{H}(pwd), n)$. Note that both hashes are necessary, as our implementation of S protects the secrecy of user passwords by storing $\mathsf{H}(pwd)$, not the passwords themselves. Furthermore, as the response requires knowledge of both n and the password (in the form of $\mathsf{H}(pwd)$), this challenge-response pair preserves the secrecy of pwd from PPT adversaries. Figure 6.2(a) shows our implementation of $\mathsf{Request}(adm, m)$, in which an administrator



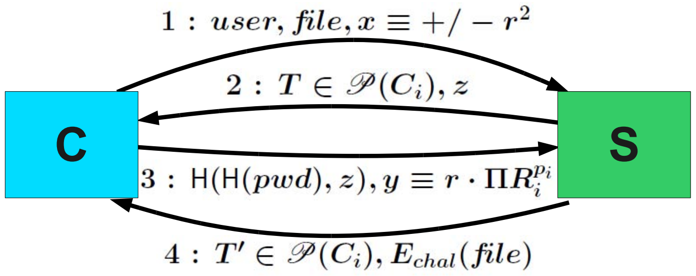
$1:\ user, m$
$2:\ n$
$3:\ \mathsf{H}(\mathsf{H}(pwd), n)$
$4:\ C_1, \ldots, C_m, N$
A
S

(a) $\mathsf{Request}(adm, m)$: Requesting a set of m challenges



$2:\ user, n, E_{otk}(C_1 : R_1^2, \ldots, C_m : R_m^2)$
$3:\ success$ or $failure$
$1:\ user, pwd, n, N, C_1, \ldots, C_m$
C
S
A

(b) $\mathsf{Enroll}(adm, pwd, C_1, \ldots, C_m, prms)$: Generating the Feige-Fiat-Shamir PUF commitments.



$1:\ user, file, x \equiv +/-r^2$
$2:\ T \in \mathscr{P}(C_i), z$
$3:\ \mathsf{H}(\mathsf{H}(pwd), z), y \equiv r \cdot \Pi R_i^{p_i}$
$4:\ T' \in \mathscr{P}(C_i), E_{chal}(file)$
C
S

(c) $\mathsf{Access}(user, file, action)$: Using Feige-Fiat-Shamir and the PUF to generate a one-time-use key to encrypt the file.

Figure 6.2. Physically restricted access control protocols. All multiplications are modulo N.

A requests a new set of challenges from the server S. The parameter N returned in step 4 is used as a modulus in the other protocols.

Our $\mathsf{Enroll}(adm, pwd, C_1, \ldots, C_m, prms)$ implementation is shown in Figure 6.2(b). Our $\mathsf{Commit}(\cdot)$ primitive consists of the pairs $(C_1, R_1^2), \ldots, (C_m, R_m^2)$, where the multiplication is modulus N. The security of this commitment relies on the intractability of computing R_i by observing $R_i^2 \pmod N$. That is, even if a PPT adversary gains access to the committed values stored on S, he can compute the modular square roots with only negligible probability, and the confidentiality of the PUF is assured. As we will explain in Section 6.5.4, we used the `mcrypt` utility to generate the cryptographic keys, thus providing the functionality of the $\mathsf{Gen}(\cdot)$ primitive.

Our instantiation of $\mathsf{Access}(user, file, action)$ is shown in Figure 6.2(c). As we mentioned previously, our choice of $\mathsf{Chal}(\cdot)$ and $\mathsf{Prove}(\cdot)$ is based on the Feige-Fiat-Shamir identification scheme. The first step of this scheme is for the prover (C) to generate a random r and send $x \equiv$ +/- $r^2 \pmod{N}$.[2] The user is then authenticated using a nonce and a cryptographic hash. Given the challenge set $T \subset \mathscr{P}(C_1, \ldots, C_m)$ (where $\mathscr{P}$ denotes the power set), C executes the PUF for each $C_i \in T$. That is, C computes $y \equiv r \cdot \prod R_i^{p_i} \pmod{N}$, where $p_i = 1$ if $C_i \in T$ and $p_i = 0$ otherwise. Thus, the proof $p \leftarrow \mathsf{Prove}(T)$ is the value +/- $y^2 \pmod{N}$. As both parties also know $\mathsf{H}(pwd)$ and the nonce z, and they can compute +/- $y^2 \pmod{N}$, they can use the proof to generate $chal \leftarrow \mathsf{Gen}(p, z, pwd)$ as required by the protocol.

There is an important subtlety here that should be noted. Under the traditional Feige-Fiat-Shamir scheme, the prover sends y and the verifier must compare both y^2 (mod N) and $-y^2$ (mod N) with the product of x and the committed values. That is, it would seem that C and S would have to attempt the encryption and/or decryption twice. However, this is not the case. S always uses $x \cdot \prod R_i^{p_i}$ (which includes the correct sign). As the decision of whether or not to flip the sign of x was made by C, C clearly knows whether the proof should be $y^2 \pmod{N}$ or $-y^2 \pmod{N}$. Hence, the encryption and decryption only need to be attempted once by each party.

In addition, readers who are familiar with existing work in generating cryptographic keys from biometrics [151] may object to our use of the responses as the secrets. In that work, the authors create a secure key $\mathcal{K}$ and compute $\Theta_{lock} = \mathcal{K} \oplus \Theta_{ref}$, where Θ_{ref} denotes the reference biometric sample. To authenticate a sample Θ_{sam} at a later point, the system applies the bit mask Θ_{lock} in an attempt to recover the key $\mathcal{K}$.

In our approach, this bit mask is unnecessary for two reasons. First, unlike biometric data, the PUF responses exist only at run-time and are never made public. In contrast, biometric data, such as fingerprints, are always present and can be harvested. Thus, PUF responses are more private and, consequently, more protected. Second, revoking a biometric is impossible; however, it must be possible to revoke the associated key. The bit mask makes this possible. In our scheme, though, revocation

[2]Randomly flipping the sign of $r^2 \pmod{N}$ ensures that the scheme is a zero-knowledge proof of knowledge.

of a PUF response R_i is simple: S stops using the associated challenge C_i. Hence, applying the bit mask to the PUF response is unnecessary for our scheme.

6.5.2 PUF Creation

We used the Xilinx Spartan-3 FPGA to implement a PUF. To simplify the circuitry, we created independent pairs of ROs, each forming a 1-bit PUF. To ensure that we could count a high number of oscillations, we implemented a 64-bit counter to receive the data from each multiplexor. Each oscillator consisted of a series of nine inverter gates. Our experiments with fewer gates resulted in the oscillator running at too high of a frequency, but nine gates offered good, consistent behavior.

We controlled the PUF execution time by incrementing a small counter until it overflowed. The Spartan-3 uses a 50 MHz clock, so a 16-bit counter overflows in approximately 1 ms. We also increased the counter size to 20 bits, which required 21 ms to overflow. We did not notice any observable difference in the consistency; hence, a 16-bit counter offers sufficient time for the oscillators to demonstrate quantifiably different behavior.

Our design is based on a 128-bit PUF. However, in our experiments, we needed to create a state machine to write the PUF result out to a serial port. The extra space for the state machine would not fit on the Spartan-3. As such, we reduced the PUF size to 64 bits for experimental evaluation. In future designs, all work will be performed on the device itself, the state machine will not be needed, and accommodating 128-bit PUFs (and larger) will certainly be feasible.

From the perspective of space on the device, the limiting factor is the usage of the look-up tables (LUTs). Implementing a 128-bit PUF on the Spartan-3 occupies 39% of the available input LUTs and 78% of slices. However, as more ROs are added, the number of slices grows only slightly, while the usage of the LUTs increases more quickly. Implementing two independent 128-bit PUFs on the same device would occupy 78% of the LUTs and 99% of slices. Note, though, that these numbers are based on our simplistic PUF design, which consists of 128 pairs of independent 1-bit ROs. More advanced designs [54] select random pairs from a pool of ROs; in such an approach, a 128-bit output can be produced from 35 ROs, whereas our approach would use 256 (128 pairs).

By implementing the full PUF as independent 1-bit PUFs, there is a direct correlation between each bit of the challenge and each bit of the response. That is, flipping only a single bit of input would result in only a single bit difference in the output. To counteract this correlation, we take a hash of the PUF output. As a result of the properties of cryptographic hash functions, a single bit difference in the PUF output will produce a very different hash result. This hash step prevents an attacker from using the one-to-one mapping to model the PUF.

6.5.3 Error Correction

PUFs are designed to be generally non-deterministic in their behavior. The physical properties of the device itself resolves this non-determinism to create a consistent and predictable challenge-response pattern. However, variations in the response are inevitable. For instance, if two ring oscillators operate at nearly identical frequencies, the PUF may alternate between identifying each as the "faster" oscillator. Reed-Solomon codes [152] correct these variations up to a pre-defined threshold.

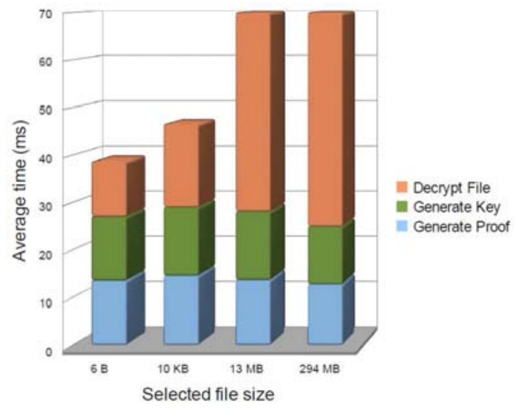
Reed-Solomon codes are linear block codes that append blocks of data with parity bits that can be used to detect and correct errors in the block. To guarantee that we can correct up to 16 bits of output for a 128-bit PUF, we use a RS(255,223) code. Note that this code operates on an array of *bytes*, rather than bits. To accommodate this, we encode each PUF output bit into a separate byte. Alternatively, we could have compacted eight bits at a time into a single byte for a more compact representation. In fact, doing so is necessary for implementations that use larger sizes of PUF output. For our current work, though, we find this encoding to be acceptable, even if it is not optimal.

RS(255,223) reads a block of 223 input symbols and can correct up to 16 errors. After converting the PUF output to a string of bytes, we pad the end of the string with 0s. The encoding produces a *syndrome* of 32 bytes that must be stored. When the PUF is executed at a later point, the response is again converted to a string of bytes and padded, and these 32 bytes are appended. The array of bytes is then decoded, correcting up to 16 errors introduced by the noisy output of the PUF.

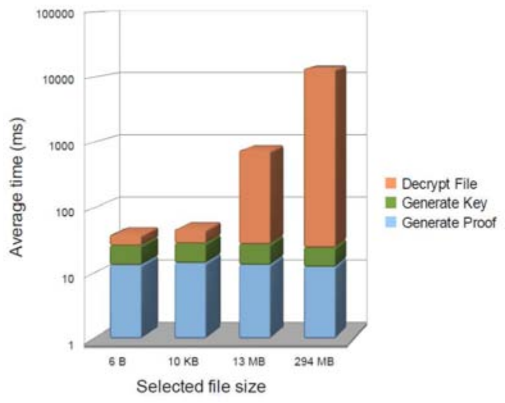
While Reed-Solomon codes can correct errors in a data block, they operate under the assumption that the original data is correct. In the case of PUFs, it is also possible that the original data varies from the normal behavior observed at later times. To counteract this initial bias, during the enrollment process, we execute the PUF three times, not once. For each bit, we do a simple majority vote. That is, the "official" PUF result is the result of the consensus of the three executions.

6.5.4 Client-Server Implementation

We implemented our protocols as a custom client-server prototype. Both applications use a custom-built package for performing arbitrary-length arithmetic operations for large numbers. All hash operations use the SHA-1 implementation by Devine [153]. We incorporated the Reed-Solomon code library created by Rockliff [154]. Recall that, in our protocols, we use symmetric key encryption in a number of steps; the symmetric keys are generated from a shared secret. In all cases, we wrote the secret to a file, used the Linux utility `mcrypt` (which reads the file and generates a strong key from the data), and immediately destroyed the file using `shred`. The cryptographic algorithm used was 128-bit AES (Rijndael). To minimize the possibility of leaking the key by writing the shared secret to a file, we used `setuid` to run server under a dedicated uid, and restricted read access to the file before writing the secret.



(a) Truncated, linear scale



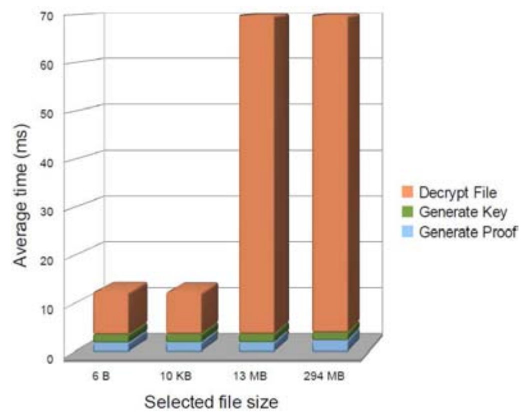
(b) Logarithmic scale

Figure 6.3. Average client-side computation time for steps 3 and 4 of the Access protocol.

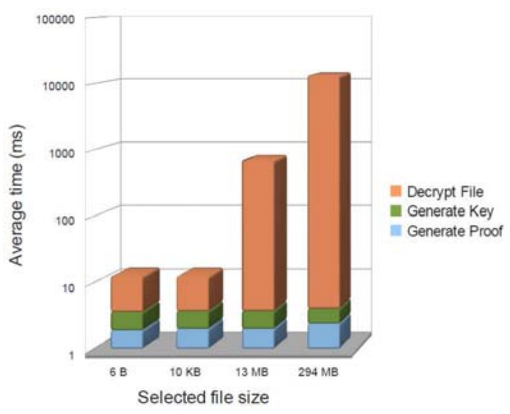
6.5.5 Storage Requirements

The storage requirements of our solution for both C and S are minimal. C must store N, the challenges C_i, and an error-correcting syndrome for each challenge. As we detailed above, N and C_i are each 128 bits, or 16 bytes in length. Each syndrome (one per challenge) is 32 bytes in length. Thus, the total storage for C in our prototype is $48m + 16$ bytes. For 16 challenges, then, the storage requirement is under 1 KB.

S also must store a minimal amount of data. S stores N and the $R_i^2 \pmod N$ commitments, each of which are 128 bits (16 bytes) in size. In addition, S stores a hash of each user's password. If SHA-1 is used, that hash is 20 bytes. If a denotes the number of devices enabled and b denotes the number of authorized users, the total storage requirement for our system is $(16m + 16)a + 20b$ bytes of data. *E.g.*, given 100 users, S can enable 1000 devices with 16 challenges each for less than 268 KB of storage.



(a) Truncated, linear scale



(b) Logarithmic scale

Figure 6.4. Average server-side computation time for steps 3 and 4 of the Access protocol.

6.6 Experimental Evaluation

We now present the experimental evaluation of of our prototype. Our evaluation goals focused on two areas. First, we strove to demonstrate that RO-based PUFs are both non-deterministic and consistent. That is, different physical instantiations of the same PUF design produce different behavior, but repeating the PUF execution on the same input and hardware produce results that can be reliably quantified as the same binary string. Our second area of evaluation was on the performance of our client-server prototype. In that portion, we show that our design offers better performance than using traditional PKI to distribute symmetric encryption keys.

The output from the PUF, implemented using a Xilinx Spartan-3 FPGA, is transferred to a client application via serial cable, although in deployed settings all operations would occur on the same device. All client and server operations were executed on a system with a 2.26GHz Intel® Core$^{\text{TM}}$ 2 Duo CPU with 3GB of 667MHz memory. The OS used was Ubuntu 9.04, with version 2.6.28-15 of the Linux kernel.

6.6.1 PUF Consistency

As noted in Section 7.4, we implemented a 64-bit PUF and wrote the serialized output to a workstation via cable. In our experiments, we observed an average of 0.2 bits that differed from the "official" PUF result. The maximum difference that we observed was 5 bits. Clearly, the use of Reed-Solomon codes that can correct up to 16 error bits at each iteration will be able to provide consistent output from the PUF, even if we double the size of the PUF to 128 bits. Furthermore, note that changes in environmental conditions, such as different temperatures, will affect the absolute speeds at which the ROs oscillate. However, the PUF result is based on the *relative* speeds; that is, increasing the temperature will slow both ROs in a pair down, but is unlikely to change which of the two oscillates faster. Consequently, the PUF shows very consistent behavior that can be used to build a reliable authentication mechanism.

6.6.2 Client/Server Performance

To evaluate the performance of our client and server implementations, we executed a series of automated file requests, given several different files sizes. In these experiments, we emulated the PUF in software. As noted in Section 6.5.2, we can control the PUF execution time; overflowing a 16-bit counter adds only 1 ms to the client computation time. Figures 6.3 and 6.4 report the amount of time for computing key portions of the Access protocol for some of the file sizes that we measured.

In these figures, "Generate Proof" (shown in blue) refers to the time to authenticate the user by generating or checking the hash $\mathsf{H}(\mathsf{H}(pwd), z)$ and the proof y sent in step 3. "Generate Key" (shown in green) refers to the amount of time required to create the 128-bit AES key needed to encrypt or decrypt the file, $E_{chal}(file)$. The AES computation is shown in orange.

Figures 6.3 and 6.4 are shown on both a (truncated) linear scale and a logarithmic scale. The key observation of these figures is that the two primary functions of our protocol, plotted as "Generate Key" and "Generate Proof," are fairly constant and minimal. The client side operations take approximately 14 ms on average, which is the same length of time as decrypting a 6-byte piece of data with AES (12 ms on average). The server burden is even less, requiring approximately 2 ms for each protocol stage and 9 ms to encrypt the file. As the file size increases, the AES encryption clearly becomes the limiting factor, as it increases approximately linearly with the file size, while our protocol overhead remains constant.

Comparing the performance of our approach with traditional PKI (specifically, RSA) required addressing a number of factors. First, the intractability assumption behind our approach (as described in the next section) states that finding the modular square root is at least as hard as factoring the product of primes, *assuming the product and the modular square are the same size.* That is, computing R_i from a 128-bit R_i^2 is only as difficult as breaking a *128-bit* RSA key, which is quite a weak claim. Thus, we needed to increase the size of the PUF output. Note, though, that the PUF execution time does not change. The only additional performance overhead is the extra time required to do the modular multiplication on larger numbers.

The other disparity between our approach and RSA is that the result of an RSA decryption would give you the key itself. In our approach, we would be left with a 1024-, 2048-, or 4096-bit value that would have to be converted into an AES key. However, based on our experiments with `mcrypt`, we observed only negligible overhead to convert this PUF output value into a key. Thus, this extra work had no measurable impact on our performance.

Figure 6.5 shows the difference in performance between our PUF-based key generation and using RSA to encrypt an AES key. The RSA modular exponentiation requires approximately four times the computation time as our client-side PUF-based key generation. Thus, our approach offers a clear performance advantage, which may be very beneficial for low-power embedded devices.

6.7 Discussion

We start this section with a brief discussion on PUF and RSA key sizes. We then focus on possible attack models for our design.

6.7.1 On Key Sizes

In the previous section, we showed the performance difference between our 128-bit PUF-based client-server architecture and various sizes of RSA keys. However, comparing the security guarantees of our system with the use of PKI to distribute symmetric keys is somewhat challenging. Revealing R_i^2 while assuming R_i to be secure relies on the assumption that computing modular square roots is intractable. [155] shows that this computation is at least as difficult as factoring the product of primes, *provided the numbers are all large.* Intuitively, though, computing a 128-bit modular square root is only as hard as factoring a 128-bit RSA key, which is quite a weak claim. We counter this criticism of our design with the following justifications.

First, attacking the R_i values in this manner can only occur at S. That is, the R_i^2 values are never transmitted in the clear where an attacker can eavesdrop. In RSA, though, public keys are used to encrypt the symmetric keys before transmitting them across the network. Transmitting keys in this manner creates an attack surface that our approach avoids.

Second, the PUF could be repeatedly polled to produce a larger output bit string. That is, appending 8 responses for a 128-bit PUF will create a 1024-bit bit string. Additionally, we showed that increasing the size yields a minimal performance cost when compared with common RSA key sizes. Consequently, we do not consider criticisms based on the key size to detract from the soundness of our overall design.

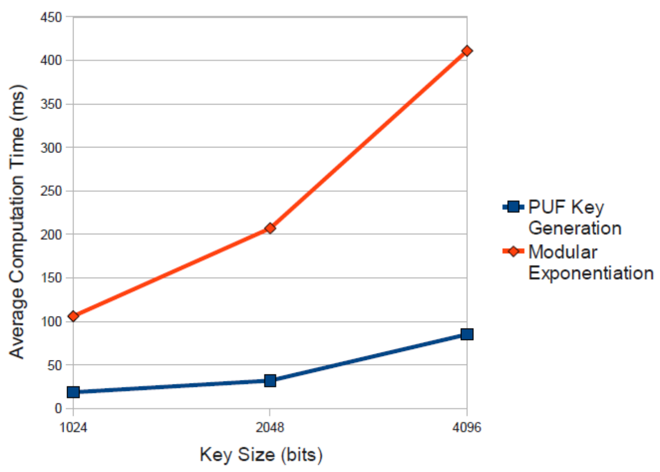


Figure 6.5. Large PUF computation compared with RSA-based modular exponentiation

6.7.2 Additional Threats & Attacks

In Section 6.4.2, we provided a formal analysis of our protocol. Here, we expand on this analysis with an informal discussion of the remaining threats to our design. First, recall that our protocol is built on the assumption that C is a trusted device. As such, we do not consider attacks in which C leaks secure data received through a legitimate access request. The presence of malware on C makes this a very realistic concern. However, we consider this threat beyond the scope of our work, and focus on what can be accomplished under the assumption that the C is trusted.

A common flaw in authentication protocols is vulnerability to a replay attack. Consider a PPT adversary $\mathcal{A}$ with a transcript of of $\mathsf{Access}(user, file, action)$, as shown in Figure 6.2(c). If either z or T were different, the replay attempt would fail. Additionally, even if both z and T are the same, $\mathcal{A}$ would learn nothing new. That is, under the PPT assumption, $\mathcal{A}$ cannot decrypt $E_{chal}(file)$. The only threat in this scenario would be if the session involved *uploading* the file from C to S. In this case, $\mathcal{A}$ could force S to revert the status of *file* to an earlier version. However, this can only happen if both z and T are identical. Assuming a large range of values for these variables, this attack can succeed with only negligible probability.

Now consider a stronger adversary $\mathcal{A}$ that has learned the pairs (C_i, R_i^2) for a particular device. Under the PPT model, such an adversary can only have learned these values by successfully attacking S. Clearly, if $\mathcal{A}$ can bypass S's protection of the pairs, he can also directly access all of the files on the system. Hence, the only remaining motivation of such an attacker is to try to model the PUF by learning the PUF responses.

The defenses against such an adversary rely on a number of factors. First, even if we set aside the PPT model and assume that the adversary has somehow learned the key used to encrypt $E_{chal}(\mathit{file})$ *and* the inputs to $\mathsf{Gen}(p, z, pwd)$. Note that this p is exactly the proof generated in the Feige-Fiat-Shamir identification scheme, which is known to be zero-knowledge. Hence, observing additional sessions provides no new information regarding the values of R_i.

Thus, $\mathcal{A}$ can only model the PUF by computing the modular square roots. Returning to the PPT model, such an attack can succeed with only negligible probability, as computing modular square roots is at least as difficult as factoring a large product of primes for composite values of N [155]. Admittedly, in our prototype, we used only 128-bit values (which is quite weak), but we demonstrated that it would be straightforward to increase the PUF output to larger sizes with minimal overhead. Hence, a PPT adversary could not model the PUF, even with possession of the pairs (C_i, R_i^2).

Finally, consider the case of a malicious administrator. Insider threats are very difficult to prevent in general, as these attackers have been granted permissions because they were deemed trustworthy. In our approach, there is no inherent mechanism for preventing a malicious administrator from enabling untrusted devices. One simple defense would be to apply separation-of-duty, thus requiring multiple administrators to input the same challenges to each device.[3] Another approach would be to require

[3]Of course, this does nothing against colluding administrators!

a supervisor to approve the enrollment request. Incorporating such defense-in-depth techniques would strengthen our scheme against these threats.

6.8 Conclusions

In this work, we have proposed a novel mechanism that uses PUFs to bind an access request to a trusted physical device. In contrast to previous work, we do not use the PUF to generate or store a cryptographic key. Rather, we incorporate the PUF challenge-response mechanism directly into our authentication and access request protocols. Furthermore, our approach avoids expensive computation, such as the modular exponentiation used in public key cryptography. As a result, our PUF-based mechanism can be used in settings where PKI or TPMs are either not available or require too much performance overhead. We have presented the details of our implementation. Our empirical results show that PUFs can be used to create a light-weight multifactor authentication that successfully binds an access request to a physical device.

7 PUF ROKS: A HARDWARE APPROACH TO READ-ONCE KEYS

In the previous chapter, we focused on the issue of defining policy restrictions based on the device used and authenticating the device based on its hardware characteristics. This chapter uses the PUF to enforce a different type of access control policy, specifically a limit on the number of times that a cryptographic key can be used. We start by discussing the notion of "read-once keys" as proposed in the cryptographic literature. We then examine how a PUF can be used to create a hardware mechanism for creating such key usage restrictions, and describe our work on such a prototype.

7.1 Read-once Keys

The term read-once keys (ROKs) describes the abstract notion that a cryptographic key can be read and used for encryption and decryption only once. While it seems intuitive that a trusted piece of software could be designed that deletes a key right after using it, such a scheme naïvely depends on the proper execution of the program. This approach could be easily circumvented by running the code within a debugging environment that halts execution of the code before the deletion occurs. That is, the notion of a ROK entails a stronger protection method wherein the process of reading the key results in its immediate destruction.

ROKs could be applied in a number of interesting scenarios. One application could be to create one-time programs [156], which could be beneficial for protecting the intellectual property of a piece of software. A potential client could download a fully functional one-time program for evaluation before committing to a purchase. A similar application would be self-destructing email. In that case, the sender could encrypt a message with a ROK; the message would then be destroyed immediately after the recipient reads the message. More generally, there is considerable interest in self-destructing data, both commercially [157] and academically [158]. In addition, the use of trusted hardware tokens have been proposed for applications including program obfuscation [159], monotonic counters [160], oblivious transfer [161], and generalized secure computation [162]. ROKs can provide the required functionality for these applications.

Another interesting application of PUF ROKs is to defend against physical attacks on cryptographic protocols. For example, consider fault injection attacks on RSA [163–167]. In these attacks, the algorithm is repeatedly executed with the same key, using a controlled fault injection technique that will yield detectable differences in the output. After enough such iterations, the attacker is able to recover the key in full. Similarly, "freezing" is another class of physical attack that can extract a key if it was *ever* stored in an accessible part of memory [168]. PUF ROKs offer a unique defense against all of these attacks because repeated execution with the same key cannot occur, and the key is *never* actually present in addressable physical memory.

The ability to generate ROKs in a controlled manner could also lead to an extension where keys can be generated and used a multiple, but limited, number of times. For example, consider the use of ROKs to encrypt a public key pk. If an identical ROK can be generated twice, the owner of pk could first use the key to create $e_{ROK}(pk)$ (indicating the encryption of pk under with the ROK). Later, an authorized party could create the ROK a second time to decrypt the key. Such a scheme could be used to delegate the authority to cryptographically sign documents.

In a sense, a ROK is an example of a program obfuscator. An obfuscator $\mathcal{O}$ takes a program $\mathcal{P}$ as input and returns $\mathcal{O}(\mathcal{P})$, which is functionally identical to $\mathcal{P}$ but indecipherable. A ROK, then, involves an obfuscator that makes only the key indecipherable. While ROKs are promising ideals, the disheartening fact is that program obfuscators–of which ROKs are one example–cannot be created through algorithmic processes alone [169]. Instead, trusted hardware is required to guarantee the immediate destruction of the key [156]. However, we are aware of no work that has specifically undertaken the task of designing and creating such trusted hardware for the purpose of generating a ROK.

In this chapter, we examine the creation of ROKs using physically unclonable functions (PUFs) [52, 53]. A PUF takes an input *challenge* $C_i \in C$, where C denotes the set of all such possible challenges. The PUF then produces a *response* $R_i \in R$, where R is the set of possible responses. The function that maps each C_i to R_i is based on the intrinsic randomness that exists in hardware and *cannot be controlled.* As such, an ideal PUF creates a mathematical function unique to each physical instance of a hardware design; even if the same design is used for two devices, it is physically impossible to make their PUFs behave identically.

Our insight for the design of such "PUF ROKs" [170, 171] is to incorporate the PUF in a feedback loop for a system-on-chip (SoC) design.[1] That is, our design is for the PUF to reside on the same chip as the processor core that performs the encryption. This integration of the PUF and the processor core protects the secrecy of the key. An attempt to read the key from memory (given physical access) will fail, because the key *never exists in addressable memory.* Also, attempts to learn the key from bus communication will be difficult or impossible, as each key is used to encrypt only a single message, and the key is *never transmitted across the bus.*

The unpredictable nature of PUFs provides a high probability that each iteration of a ROK generation will produce a unique, seemingly random key. Yet, to ensure that a key can be generated to perform both encryption and decryption, the PUF must be initialized repeatedly to some state, thus providing the same sequence of keys. To accomplish this, Alice could provide an initial seed to produce a sequence of keys that are used to encrypt a set of secrets. Alice could then reset the seed value before making the device available to Bob. Bob, then, could use the PUF to recreate the keys in order, decrypting the secrets. As Bob has no knowledge of the seed value, he is unable to reset the device and cannot recreate the key just used.

[1]Our design could also be made to work for application-specific integrated circuits (ASICs), but we limit our discussion to SoC designs for simplicity.

Astute readers will note the similarities between our approach and using a chain of cryptographic hashes to generate keys. That is, given a seed x_0, the keys would be $\mathsf{H}(x_0)$, $\mathsf{H}(\mathsf{H}(x_0))$, etc., where H denotes a cryptographic hash function. The insight of our approach is that a PUF, as a trusted piece of hardware, can provide a hardware-based implementation that is analogous to a hash function, but is more secure than software implementations of such algorithms.

7.2 ROK Formalisms

Our formal notion of a ROK is based on an adaptation of Turing machines. Specifically, define the machine T to be

$$T =< Q, q_0, \delta, \Gamma, \iota >$$

where Q is the set of possible states, q_0 is the initial state, δ defines the transition from one state to another based on processing the symbols Γ, given input ι. Readers familiar with Turing machines will note that ι is new. In essence, we are dividing the traditional input symbols into code (Γ) and data (ι). For the sake of simplicity, we assume that ι only consists of messages to be encrypted or decrypted and ignore other types of input data. Thus, the definition of δ is determined by the execution of instructions $\gamma_1, \gamma_2, \ldots, \gamma_i$, where consuming $\gamma_i \in \Gamma$ results in the transition from state q_i to q_{i+1}. Based on this formalism, we propose the following primitives.

- The **encrypt primitive** $\mathsf{Enc}(\gamma_i, q_i, m)$ encrypts the message $m \in \iota$ given the instruction γ_i and the state q_i. The system then transitions to q_{i+1} and produces the returned value as $e(m)$ as a side effect.
- The **decrypt primitive** $\mathsf{Dec}(\gamma_j, q_j, e)$ decrypts the ciphertext $e \in \iota$ given the instruction γ_j and the state q_j. If the decryption is successful, the primitive returns m. Otherwise, the return value is denoted $\emptyset$. The system then transitions to q_{j+1}.

Informally, γ_i and q_i describe the current instruction and the contents of memory for a single execution of a program, and capture the state of the system just before executing the encrypt or decrypt primitive. That is, if the execution of the program is suspended for a brief time, γ_i, q_i would describe a snapshot of the stack, the value stored in the instruction pointer (IP) register, the values of all dynamically allocated variables (*i.e.,* those on the heap), etc. In short, it would contain the full software image for that process for that precise moment in time. Once the program is resumed, the symbol γ_i would be consumed, and the system would transition to state q_{i+1}. Given these primitives, we present the following definition.

Definition: A **read-once key** (ROK) is a cryptographic key $\mathcal{K}$ subject to the following conditions:

- Each execution of $\mathsf{Enc}(\gamma_i, q_i, m)$ generates a new $\mathcal{K}$ and yields a transition to a unique q_{i+1}.
- The first execution of $\mathsf{Dec}(\gamma_j, q_j, e)$ returns m and transitions to q_{j+1}. All subsequent executions return $\emptyset$ and transitions to q'_{j+1}, even when executing the machine $< Q, q_0, \delta, \Gamma, \iota >$ with e, except with negligible probability.
- The probability of successfully decrypting e without the primitive $\mathsf{Dec}(\gamma_j, q_j, e)$ is less than or equal to a security parameter ϵ $(0 < \epsilon < 1)$, even when given identical initial states. ϵ must be no smaller than the probability of a successful attack on the cryptographic algorithms themselves.

What these definitions say is that the ROK Turing machine is non-deterministic. Specifically, during the first execution of a program[2] that encrypts a message m, δ will define a transition from q_i to q_{i+1} based on the primitive $\mathsf{Enc}(\gamma_i, q_i, m)$. However, the second time, the key will be different, and the state transition will be from q_i to

[2]Observe that the program doing the encryption is separate from the one doing the decryption. If the encryption and decryption occurred in the same program, the decryption would succeed, as the key would have just been dynamically generated. In contrast, when the programs are distinct, only the first execution of the decryption program will succeed.

q'_{i+1}. Similarly, the first execution of a program that decrypts $e(m)$ will traverse the states $q_0, \ldots, q_j, q_{j+1}$, where q_{j+1} is the state that results from a successful decryption. However, returning the machine to its initial state q_0, using the same instructions Γ, the state traversal will be $q_0, \ldots, q_j, q'_{j+1} = q_{j+1}$, because the decryption fails. Thus, ROKs incorporate some unpredictable element that does not exist in traditional Turing machines: the history of prior machine executions. That is, for any given machine T, only the first execution (assuming either the encrypt or decrypt primitive is executed) will use the transitions defined by δ. The second (and subsequent) executions will use δ', as the state after the primitive is invoked will differ.

Clearly, these definitions capture the intuitive notion of a ROK. The key $\mathcal{K}$ is generated in an on-demand fashion in order to encrypt a message. Later, $\mathcal{K}$ can be used to decrypt the message, but only once. After the first decryption, the key is obliterated in some manner. Specifically, even if the contents of memory are returned to match the program state γ_j, q_j as it existed before the first call to $\mathsf{Dec}(\gamma_j, q_j, e)$, the decryption will fail. The intuition here is that a special-purpose hardware structure must provide this self-destructing property.

Observe that an adversary $\mathcal{A}$ may opt to attack the cryptographic algorithms themselves. In such an attack, the number of times the key $\mathcal{K}$ can be read by an authorized party is irrelevant: $\mathcal{A}$ is never authorized. If the cryptographic scheme is sufficiently weak, $\mathcal{A}$ may succeed in recovering the message (or the key itself). The ROK property offers no additional security against such an attack. That is, we are making no special claims of cryptographic prowess. For this reason, we require that ϵ be no smaller than the probability of a successful attack on the cryptographic scheme employed.

What is unique about our technique is that we are offering a means to limit the usage of a key by an authorized party. Clearly, with sufficient motivation, this authorized party may become an adversary himself, attempting to recover the key $\mathcal{K}$ and subvert the system. The parameter ϵ offers a means to specify the system's defense against such an insider threat. For the most sensitive data, an implementation of our design could require a very low level of ϵ, making the probability of subverting the ROK property equal to the probability of a brute-force attack on the cryptographic algorithm. In applications that are less sensitive (*i.e.,* the ROK property is desirable, but not critically important), ϵ could be larger. In short, ϵ captures the flexibility to adjust the security guarantees of the ROK according to desired implementation characteristics. We will explore this idea more in Sections 7.3 and 7.5.

7.3 PUF ROKs

In this section, we propose the use of PUFs to generate ROKs, which we call PUF ROKs. Like previous work [49], our design is based on the idea of using the PUF output to generate a transient key dynamically. We start this section by describing the basic hardware architecture for creating a PUF ROK component. We then proceed to prove formally that this architecture captures the desired ROK characteristics. This section concludes with descriptions of how PUF ROKs can be used in both symmetric key and public key cryptography.

7.3.1 PUF ROK Overview

The high-level view of our hardware architecture for generating PUF ROKs consists of a number of components. We formally define these components and their functional connections as follows.

- The **processor core** (PC) executes the desired application. The PC has access to volatile random access memory (RAM) for implementing the typical C-language execution constructs, such as a stack and heap. The PC contains an interface to a physically distinct **crypto core** (CC).
- The CC is a stand-alone hardware component that provides cryptographic services to the PC. The CC provides the following service interface to the PC:

  - $\mathsf{Init}(x_0)$: an initialization routine that takes an input x_0 as a seed value for the PUF. There is no return value.
  - $\mathsf{Enc}(m)$: an encryption primitive that takes a message m as input and returns the encrypted value $e(m)$.
  - $\mathsf{Dec}(e(m))$: a decryption primitive that takes a ciphertext as input. Given $e(m)$ repeatedly, this service returns the plaintext m only on the first execution. Subsequent calls to this service for $e(m)$ return $\emptyset$.

- The CC has a unidirectional interface with a **register** (Reg). Whenever the CC's $\mathsf{Init}(x_0)$ service is invoked, the CC writes x_0 (or a value derived from x_0, if so desired) into Reg.

- The CC can poll the **PUF**. When this occurs, the value stored in Reg is used as the PUF challenge. The response is then fed into an **error correction unit** (ECU). After performing mode-specific functions, the ECU returns a sanitized PUF output to the CC, while simultaneously overwriting the contents of Reg. When decrypting, the write back to Reg is contingent on feedback from CC. That is, Reg would only be overwritten during $\mathsf{Dec}(e(m))$ if the decryption was successful.

Figure 7.1 shows a high-level view of a SoC implementation of our PUF ROK design. The key insight of this approach is the the PUF-ECU-Reg portion form a feedback loop. The PUF uses the values stored in the Reg as its input challenge C_i. The raw response R_i is run through an error correction routine to produce R_i', which is written back into the Reg. The cleaned response is also reported to the CC for use in the cryptographic operations.

The operation of the ECU depends on the cryptographic primitive invoked. In the case of encryption, the key $\mathcal{K}$ is just being created. As such, there are no errors to correct. Instead, the ECU uses the raw PUF output as the "correct" value and generates a small amount of error-correcting data. This data is stored in a local cache

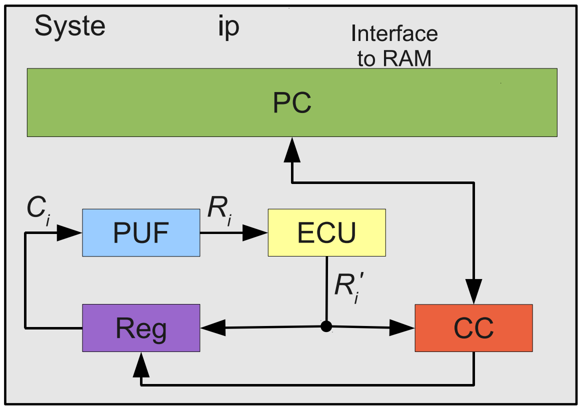


Figure 7.1. Components for a SoC PUF ROK design

that is accessible only to the ECU itself. Later, when decryption occurs, this data is used to correct any transient bit errors that may have occurred during the PUF execution. Observe that, as the error correction occurs *before* the response is stored in the Reg, this design structure ensures that the challenge inputs are correct.

The security parameter ϵ is used to specify the size of x_0. Specifically, to meet the security guarantees dictated by the ROK formalism, x_0 must be at least $\lceil -\log_2 \epsilon \rceil$ bits[3] in length, with each bit distributed uniformly. For completeness, we assume that x_0 has a length of at least one bit (for the cases when $\epsilon > 1/2$). Our subsequent analysis holds, but including this fact in later proofs adds unnecessary mess to the notation. As such, we will omit this fact and implicitly assume that x_0 is at least one bit in length.

In this architecture, we make two simplifying assumptions. First, we assume that the challenges and responses are the same length. We also assume that Reg consists of a small storage cache of the same size.[4] In implementations where these assumptions do not hold, additional hardware components may be required.

[3]Recall that $0 < \epsilon < 1$. As a result, $\log_2 \epsilon < 0$, so $\lceil -\log_2 \epsilon \rceil$ is a nonnegative integer.
[4]Depending on the size of the PUF output, Reg may correspond to an array of hardware registers. *E.g.,* if the PUF output is 256 bits and the hardware registers store only 32 bits each, then Reg consists of eight physical registers.

7.3.2 System Details

Our architecture, as previously proposed, seems rather limited in use. Primarily, keys must be used for decryption in the same order as they are created. For example, consider the case where two messages m_1 and m_2 are encrypted. The first encryption would generate $\mathcal{K}_1$ and the second creates $\mathcal{K}_2$. To switch to decrypt mode, the $\mathsf{Init}(x_0)$ primitive would be required to return the Reg to its original state. The implication of this design is that the user must perform $\mathsf{Dec}(e(m_1))$ before $\mathsf{Dec}(e(m_2))$.

An intuitive solution would be to pass a counter n along with the $\mathsf{Dec}(e(M))$ invocation, indicating that the PUF must be polled n times before the appropriate key would be reached. Hence, to decrypt the second message first, the invocation would be $\mathsf{Dec}(e(m_2), 2)$. This solution is problematic, though. Specifically, once $\mathsf{Dec}(e(m_1), 2)$ is invoked, the contents of Reg would no longer contain x_0, and there would be no way for $\mathsf{Dec}(e(m_1), 1)$ to generate $\mathcal{K}_1$.

A similar problem is that any switch between encryption and decryption would require resetting the contents of Reg and polling the PUF multiple times. For instance, assume the user has encrypted three messages m_1, m_2, and m_3. The PUF ROK would have generated keys $\mathcal{K}_1, \mathcal{K}_2$, and $\mathcal{K}_3$. To decrypt $e(m_1)$, $\mathsf{Init}(x_0)$ restores the Reg to its initial state, and $\mathsf{Dec}(e(m_1))$ is invoked. After the decryption, Reg is storing R_1. In order to encrypt message m_4, the PUF would need to be polled two more times to ensure that key $\mathcal{K}_4$ is generated. This can become very complicated to maintain.

To address these problems, we expand the details of our design as shown in Figure 7.2. In this figure, we partition the high-level Reg into distinct registers for processing the challenge input for encryption (EncReg) and for decryption (DecReg), as well as one that stores the seed value (SeedReg). We also introduce an error-correcting cache (EC Cache). The intuition in this design is that the ECU will store n error-correcting codes that can be accessed in arbitrary order. Once the first k codes have been used, they can be replaced. When this happens, the ECU synchronizes with the SeedReg (as indicated by the Sync line).

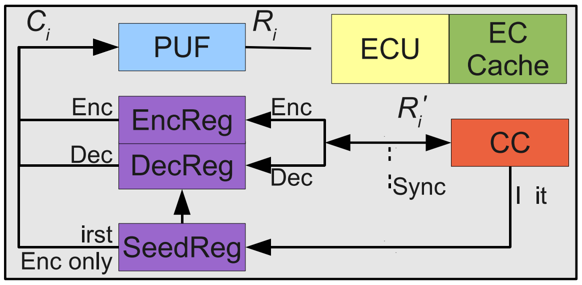


Figure 7.2. Extension components for out-of-order PUF ROKS

The operation of our PUF ROK architecture can be illustrated by the following example. Assume $n = 4$. A sample work flow could be as follows:

1. The user initializes the system with $\mathsf{Init}(x_0)$.

2. Three messages, m_1, m_2, and m_3 are encrypted. The keys $\mathcal{K}_1, \mathcal{K}_2$, and $\mathcal{K}_3$ are derived from the PUF responses R_1, R_2, and R_3, respectively. The ECU stores error correcting codes EC_1, EC_2, and EC_3 in its cache.

3. Message m_2 is decrypted by invoking $\mathsf{Dec}(e(m_2), 2)$. The contents of SeedReg are copied into DecReg, and the PUF is polled twice to generate R_2 and the corresponding $\mathcal{K}_2$. The ECU marks EC_2 as invalid, assuming the decryption is successful.

4. Message m_4 is encrypted, using R_4 and $\mathcal{K}_4$. EC_4 is generated and stored in the cache.

5. Message m_1 is decrypted by $\mathsf{Dec}(e(m_1), 1)$. At this point, as both EC_1 and EC_2 have become invalid, the ECU initiates the Sync action.

6. During the Sync, the ECU takes control of the PUF feedback loop. The PUF is polled twice, using the contents of SeedReg as the challenge input (x_0). As a result, responses R_1 and R_2 are generated, and R_2 is ultimately stored in

SeedReg. As a result of Sync, the part of EC Cache that was used to store EC_1 and EC_2 is now marked as available.

7. To encrypt m_5, $\mathsf{Enc}(m_5)$ is invoked like normal, using the contents of EncReg as the challenge input. The corresponding EC_5 is then stored in one of the newly available slots in EC Cache.

While this approach addresses the complication of using keys out of order, a simple extension makes the design even more powerful. Consider the case where a key is needed n times, rather than just once. *E.g.*, if Alice needs Bob to encrypt 10 messages on her behalf, she could either use 10 ROKs or she could employ, for lack of a better term, a read-10-times-key. The extension above, with the integrated EC Cache, could accommodate this request by storing the EC_i codes 10 times. The codes would then become invalid after all 10 slots in the EC Cache have been marked as such.

7.3.3 Formal Proof of PUF ROK

While it may seem intuitive that our architectural design captures the essence of a ROK, we present the following formal proof to illustrate the use of the security parameter ϵ. In this theorem, we use the notation $|s|$ to denote the length of the bit string s. To simplify the proof, we elide the details discussed in Section 7.3.2, and simply use the high-level architectural terms from Section 7.3.1.

Theorem. Assuming an ideal PUF and adequate error correction are employed, the PUF ROK architectural design successfully enforces the ROK criteria.

Proof. To demonstrate the claim, we must show that all three properties of ROKs hold. Clearly, our definitions of the $\mathsf{Enc}(m)$ and $\mathsf{Dec}(e(m))$ services provided by the CC match those of the ROK definition. However, to see that the first two ROK properties hold, we must consider the interaction of the components. When the

$\mathsf{Enc}(m)$ service is invoked, the contents of Reg are used as input to the PUF and replaced with the output. This output is also used to generate the key $\mathcal{K}$. Thus, the first property holds.

To switch the PUF ROK to decrypt mode, the seed x_0 must again be written to Reg. This action must be done by the encrypting party. After this is done, the $\mathsf{Dec}(e(m))$ primitive will involve polling the PUF, thus using x_0 as the PUF's input challenge once again. The ECU ensures that the new PUF result matches the previous output, and the decryption key $\mathcal{K}$ will be identical to that used for encryption. However, the new PUF result also replaces x_0 in the Reg. Hence, as the PUF is assumed to be ideal, the first execution of $\mathsf{Dec}(e(m))$ will produce the correct key $\mathcal{K}$. Any future execution of $\mathsf{Dec}(e(m))$ will, with near certainty, produce an incorrect key; the CC will then return $\emptyset$. Thus, the ideal PUF assumption ensures that the second criterion holds.

To complete the proof, we must show that our design satisfies the third criterion, which is that the probability of bypassing the restriction on the $\mathsf{Dec}(\gamma_j, q_j, e)$ primitive is less than or equal to ϵ. There are three ways this could occur. First, the adversary $\mathcal{A}$ could succeed in an attack on the cryptographic scheme itself. However, by definition, the probability of this occurring is guaranteed to be no more than ϵ, and the third criterion would hold. The second possibility would be for the ECU to produce the key $\mathcal{K}$ erroneously. However, this violates our assumption that adequate error correction is employed. Hence, this case cannot occur.

The third case is for a later execution of the PUF to produce x_0 as a result. That is, there must exist some j such that the contents of Reg would be the sequence

$$x_0, R_1, R_2, R_3, \ldots, R_j = x_0$$

Recall that we assumed $(m \geq 1)$

$$|C_j| = |R_j| = |R'_j| = |x_0| = m \geq \lceil -\log_2 \epsilon \rceil$$

Then the probability that any $R_j = x_0$ would be $1/2^m$, which gives us

$$P(R_j = x_0) = \frac{1}{2^m} \leq \frac{1}{2^{\lceil -\log_2 \epsilon \rceil}} \leq \frac{1}{2^{\log_2 \epsilon^{-1}}} = \frac{1}{\epsilon^{-1}} = \epsilon$$

That is, the probability that any polling of the PUF will produce an output that matches the seed x_0 is less than the security parameter ϵ. Hence, the third required property of ROKs also holds. Thus, the PUF ROK architecture successfully implements the ROK requirements under the ideal PUF assumption. $\square$

7.3.4 Symmetric Key PUF ROKs

The functionality of the CC depends on the cryptographic application. In the preceding discussion, we were focusing on a symmetric key application, in essence. However, there are a few more details regarding the operation of the CC that we must address here.

As noted above, in symmetric key cryptographic applications, the PC issues the command $\mathsf{Enc}(m)$, where m indicates the message to be encrypted. The CC then issues the command $\mathsf{Init}(x_0)$, which writes the value x_0 into the Reg. The PUF then uses x_0 as C_1 and generates R_1, which is passed to the ECU. The ECU then generates the necessary error-correcting codes EC_1 to ensure the key is later recoverable, even if the noisy output of the PUF produces a small number of bit errors.

Next, to guarantee a strong key from R_1, the CC applies a cryptographic hash. That is, to generate a 256-bit AES key, the CC computes $\mathcal{K}_1 = \mathsf{H}(R_1)$, where H is the SHA-256 hash function. While an ideal PUF is assumed to produce random mappings, we employ the hash function in this way to add to the entropy of the system. That is, if R_i and R_j (the responses produced by two different PUF pollings) differ by only a single bit, $\mathsf{H}(R_i)$ and $\mathsf{H}(R_j)$ will have a Hamming distance of 128 bits on average. As a result, even if an attacker is able to recover the key just by observing the plaintext and ciphertext, the hash prevents modeling the PUF, as doing so would require the attack to create a pre-image of the hash.

Once the CC has polled the PUF and generated the key, the encrypted message $e(m)$ is provided to the PC. Later, when the recipient wishes to decrypt the message (which can only be done once), the PC issues the command $\mathsf{Dec}(e(m))$ to the CC. The CC then resets the Reg with $\mathsf{Init}(x_0)$, and polls the PUF to recreate the key. The decrypted message, then, is returned to the PC.

For the sake of completeness, observe that we have never detailed how x_0 is determined. One approach, which depends only on the device itself, would be to take the timestamp ts when the PC invokes the Init primitive, and uses $x_0 = \mathsf{H}(ts)$. In another approach, the PC could use a user's password, and hash it similarly. Thus, the seed value can be determined in multiple ways.

7.3.5 Public Key PUF ROKs

Now consider the case of public key cryptography. In this setting, we start with the assumption that the CC contains the necessary parameters for the public key computations. For instance, if the RSA cryptosystem is used, the CC knows (or can create) the two large prime numbers p and q such that $n = pq$.[5] The goal, then, is to generate a pair (pk, sk), where pk denotes a public key and sk denotes the corresponding private key.

In contrast to the symmetric key approach, the CC does not need to generate the ROK twice. As such, the $\mathsf{Init}(x_0)$ function is optional. However, the CC still polls the PUF to generate the pair of keys. The challenge with using a PUF to create a public key pair, though, is how to generate a bit string that is long enough. A strong RSA key, for example, is at least 2048 bits long. But creating a 2048-bit PUF output would require a prohibitively large circuit design.

Instead, our approach is for the CC to buffer a series of PUF results. For instance, if the PUF produces a 256-bit output, the CC could use R_i as bits 0-255, R_i+1 as bits

[5]Readers familiar with the properties of RSA will observe that it is necessary for the CC to create and store the primes p and q securely. Such functionality is common in existing cryptographic processors, such as a TPM. Consequently, we assume that the CC is designed to ensure the secrecy of p and q is preserved, and omit further discussion of this matter.

255-511, and so forth. Once the CC has polled the PUF to get a sufficient number of random bits, the next challenge is to convert this bit string into a strong key. For simplicity, we assume the use of RSA.

Let e denote the candidate key that the CC has received from the PUF. In order to use e as an RSA key, e must be coprime to $\phi(n) = (p1)(q1)$. By applying the Euclidean algorithm, the CC can compute the greatest common divisor $g = gcd(e, \phi(n))$. If $g = 1$, e and $\phi(n)$ are coprime, and e can be used as is. Otherwise, $e' = e/g$ can be used. The secret key sk, then becomes e or e' as appropriate. To compute the public key pk, the CC computes the modular multiplicative inverse of sk by using the extended Euclidean algorithm. That is, the CC computes d such that $sk \cdot d \equiv 1 \pmod{\phi(n)}$. This value d then becomes the public key pk. Given this key pair (pk, sk), the PUF ROK can be used by the PC in multiple ways. First, the PC could issue the command $\mathsf{Sign}(m)$ to the CC, requesting a cryptographic signature. After generating (pk, sk), the CC uses sk to sign m, returning the signature and the public key pk to PC. pk can then be used by a third party to verify the signature.

Alternatively, the PC could issue the command Gen, which tells the CC to generate the key pair. Instead of using the keys immediately, the CC stores sk and returns pk to the PC. A third party wishing to send an encrypted message to the PC could use pk as needed. Then, the PC would issue $\mathsf{Dec}(m)$ to have the CC decrypt the message. While this violates the spirit of the ROK principle (as sk would need stored somewhere), sk could simply be thrown away during the Gen procedure. Later, when the decryption occurs, the sk would be recreated, making the public key PUF ROK work similarly to the symmetric key version.

Finally, consider the case where the third party needs assurance that the public key pk did, in fact, origin from the PUF ROK. This can be accomplished if the CC contains a persistent public key pair, similar to the Endorsement Key (EK) stored in a Trusted Platform Module (TPM). In addition to providing the pk to the PC, the CC could also return $\mathsf{Sign}_{EK}(pk)$, denoting the signature of the pk under this persistent key. This technique provides the necessary assurance, as the persistent key is bound to the CC. However, this requires a key management infrastructure similar to existing TPM-based attestation schemes.

7.3.6 Practicality and Applications

One obvious objection to PUF ROKs is that they assume shared access to the resource. In many instances, this assumption does not hold. However, as we will describe in Section 7.4, we have implemented a proof-of-concept PUF ROK on a small portable device that measures 44×60 mm. Based on this experience, we find that it would be quite reasonable to integrate PUF ROK functionality into devices similar to USB thumb drives. Clearly, such a device could be passed between users who are generally in close proximity.

For remote users, a more complicated structure would be needed. Specifically, a PUF ROK for remote use would function in a manner similar to a TPM. That is, Bob's TPM-like PUF ROK would generate a public key that Alice would use for encryption. Later, Bob's device would generate the corresponding private key to decrypt the message. Clearly, Alice would need assurance that the public key actually came from a PUF ROK. Unfortunately, there is no easy solution to this. Instead, we find the most straightforward approach to be exactly that used by TPMs. That is, the PUF ROK device would require a certificate created by the manufacturer, storing a persistent private key generated by the manufacturer. This key would then be used to sign all PUF ROKs from that device, and Alice could confirm the signature with the manufacturer's public key.

In short, our PUF ROK device infrastructure for remote users would mirror TPM behavior. However, there is one major exception: The device is trusted to enforce the behavior that the PUF ROK can only be used once. This behavior does not exist in TPMs. However, PUF ROKs could clearly be integrated into any custom SoC design that is functionally similar to a TPM.

Now consider the applications of PUF ROKs. Goldwasser *et al.* [156] proposed a technique for one-time programs, based on the assumption that the application is encrypted under a one-time use key. Closely related to one-time programs are delegated signatures. If Alice has a persistent key sk_A, she could encrypt this key with the PUF ROK as $e(sk_A)$. Bob would then provide this encrypted key to the PUF ROK, which decrypts it and uses the decrypted key to sign a single document on Alice's behalf.

PUF ROKs could also be used to generate self-destructing messages. If Alice has a portable PUF ROK device, she could use it to generate $\mathsf{Enc}(m)$. After receiving the message, Bob could use the device to decrypt the message. Once this is done, repeated attempts to decrypt the message would fail, as the Reg would no longer store the necessary challenge input.

Finally, consider the scenario of usage control. In this case, Bob has a public key PUF ROK device that contains the TPM-like endorsement key EK. Bob could use the device to retrieve the signed pk, which he sends to Alice. Alice, after confirming the signature, uses the key to encrypt the protected resource, sending the result to Bob. Bob can then use the sk stored on the PUF ROK to access the resource. Once the CC uses sk, this key is no longer accessible, and access to the resource is revoked.

7.4 Implementation

To demonstrate a proof-of-concept for our PUF ROK design, we developed a prototype implementation. As the design requires a combination of hardware and software, we desired a platform that would be advantageous for both pieces of development. Our solution was to use the KNJN Saxo-L development board [172]. This board features an Altera Cyclone EP1C3T100 field-programmable gate array (FPGA) alongside an NXP LPC2132 ARM processor. The FPGA and ARM are directly connected via a Serial Peripheral Interface (SPI). The board also offers a USB-2 adaptor, in addition to the JTAG adaptor that is commonly used for FPGA development. In addition, the form factor of the board measures 44×60 mm, making it highly portable.

In our prototype development, we chose to use the FPGA only for components that require a hardware implementation. Specifically, we implemented a small PUF and register on the FPGA to capture the essence of the feedback loop. All other portions, including the error-correcting unit (ECU) and the crypto core (CC) were implemented in software on the ARM processor. Figure 7.3 shows the high-level layout of our implementation on the KNJN board.

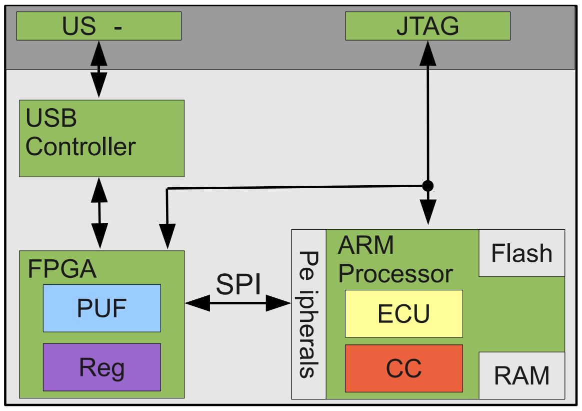


Figure 7.3. Basic hardware layout of a PUF ROK implemented on the KNJN Saxo-L development board

Our PUF design consisted of 32 1-bit ring oscillator PUFs, as shown in Figure 6.1. Each of these circuits consisted of a ring oscillator constructed from 37 inverting gates. In our experiments, we found that using fewer than 37 gates yielded less consistency in the PUF behavior. That is, smaller PUFs increase the number of bit errors that must be corrected. The output from the ring oscillators was linked to 20-bit counters that were controlled by a 16-bit timer. The timer was synchronized with a 24 MHz clock, indicating that the timer would expire (as a result of an overflow) after 2.73 ms. When the timer expires, the values in the counters are compared, producing a 1 or 0 depending on which counter had the higher value. This design used 2060 of the 2910 (71%) logic cells available on the FPGA. Each execution of the PUF produced

32 bits of output. Consequently, to generate larger keys, the ARM processor polled the PUF multiple times, caching the result until the key size was met.

To put the performance of the PUF into perspective, we compared the execution time with measurements [173] reported by NXP, the device manufacturer. Some of NXP's measurements are reported in Figure 7.1. As each PUF execution (producing 32 bits of output) requires 2.73 ms to overflow the timer, it is slower than encrypting one kB of data in AES. Observe, though, that larger PUFs would still only require 2.73 ms. Consequently, the overhead of executing the PUF can remain small, especially as large amounts of data are encrypted or decrypted.

Table 7.1
NXP cryptographic measurements

Symmetric Algorithm	Time (ms/kB)	RSA Operation	Time (s)
AES-CBC	1.21	1024-bit encrypt	0.01
AES-ECB	1.14	1024-bit decrypt	0.27
3DES-CBC	3.07	2048-bit encrypt	0.05
3DES-ECB	3.00	2048-bit decrypt	2.13

The comparison the RSA encryption and decryption is stark. Observe that the 2.73 ms required to execute the PUF is 27.3% of the time to perform a 1024-bit encryption in RSA. As the key size increases (assuming the PUF size is increased accordingly so that only one polling is needed), the PUF execution time becomes 0.13% overhead for 2048-bit RSA decryption. Thus, the performance impact of polling the PUF during key generation is minimal.[6]

Regardless of the size of the PUF used, one challenge that is unavoidable is random inconsistencies in the PUF output. Specifically, a small number of bit errors will inevitably occur as a result of the randomness inherent to PUFs. To counteract this

[6]Obviously, there is additional work required to convert the PUF output into a usable key. However, the precise timing of this work is implementation-dependent, and the algorithms typically employed are significantly more efficient than the modular exponentiation. As such, we focus solely on the PUF measurement in our analysis.

behavior, we employed Reed-Solomon (RS) error-correcting codes [174]. Specifically, we adapted the Rockliff's implementation [154] in our prototype.

RS(n, k) codes operate as follows. A string of k symbols, each m bits in length, is concatenated with an $n - k$ *syndrome*, where $n = 2^m - 1$. Based on this syndrome, when the input is used again, the codes can correct up to $(n-k)/2$ corrupted symbols. In our implementation, we used RS$(15, 9)$ codes, as the PUF output each time was 32 bits. Observe that $m = 4$ in this code, so this $k = 9$ is a sufficient size, as there would be nine 4-bit symbols (a total of 32 bits). Furthermore, when the PUF is polled later, this code can account for three corrupted symbols (potentially up to 12 bits in the PUF). However, in our experiments, the average Hamming distance between the original PUF response and later responses was 0.1, with a maximum of two bit errors that occurred in one execution. Clearly, this code is sufficient for our implementation.

In our implementation, we adapted the PolarSSL cryptographic library [175] for execution on the ARM processor. This open source library is designed to be small enough for use in embedded devices. The LPC2132 model offers only 16 kB of RAM and 64 kB of Flash memory. As this is quite a small amount of storage space, the library actually would not fit in the device's available memory. Specifically, the library contains a number of I/O functions that are not suited for our implementation. Consequently, we customized the code by removing this unused functionality and were able to make the code fit within the confines of the device.

Based on the experience of building this prototype, we offer the following insights for creating production-quality PUF ROKs. First, employing both an FPGA and an ARM processor adds to the complexity of the system design. As an alternative approach, one could leverage ARM softcore designs and place them within the FPGA itself. This would simplify the circuitry on the board itself.

Additionally, our current design is not optimal for produc- tion-quality PUFs. Specifically, it creates a one-to-one correlation between a single bit in the input challenge C_i and the corresponding response bit in R_i. As such, if C_i and C_j differ by only a single flipped bit, then R_i and R_j will also differ by only the same flipped bit.

To prevent this correlation and produce behavior that more closely resembles an ideal PUF, the circuit design should randomly select pairs from a pool of ring oscillators, rather than having persistent pairs. As this issue was addressed in [54], interested readers should consult that work for more information.

Finally, our work used a resource-limited development board. Specifically, 2910 logic cells is considerably smaller than most FPGAs (*e.g.,* the Xilinx Spartan-3E has 10,476). Also, recall that the amount of memory available was too small to hold a cryptographic library that is *intended* for embedded devices. Consequently, we feel confident that our architecture could be easily adapted to larger devices, even as if the size of the PUF is increased to produce larger keys.

7.5 Security Analysis

For our security analysis, we consider the case of a probabilistic polynomial-time (PPT) attacker $\mathcal{A}$, with two goals. First, the goal of $\mathcal{A}$ is to recover just the key used to encrypt or decrypt a single message. The second goal considered is to model the PUF, which would enable the attacker to emulate the PUF ROK in software, thereby negating the hardware ROK guarantee. Initially, in both cases, we assume the adversary is capable of (at most) eavesdropping on bus communication. That is, the adversary is unable to observe communication between the cores in the SoC design.

Under this model, $\mathcal{A}$ is able to observe the data passing between the PC and memory, or between the PC and a network. Observe, though, that these messages consist exclusively of the plaintext m and the encrypted $e(m)$. Thus, the attack is a known-plaintext attack. However, this information offers no additional knowledge to $\mathcal{A}$. Even if $\mathcal{A}$ managed to reconstruct the key $\mathcal{K}$ (with negligible probability under the PPT model), this key is never used again.

The only use of reconstructing $\mathcal{K}$ in this manner is to attempt to reverse engineer the behavior of the PUF. However, recall that our design involved hashing the PUF output when creating the keys. Consequently, $\mathcal{K} = \mathsf{H}(R_i)$, where H is a robust cryptographic hash function. As a result, $\mathcal{A}$ again has only a negligible probability of reconstructing R_i. Yet, we can take this analysis even further, because R_i by itself is useless. That is, $\mathcal{A}$ would also need to know the corresponding C_i (or R_{i+1}) to begin to model the PUF. Thus, $\mathcal{A}$ would have to accomplish a minimum of *four* feats, each of which has only a negligible probability of occurring. Thus, we do not find such an attack to be feasible.

To continue the analysis, we loosen our assumed restrictions and grant $\mathcal{A}$ the ability to probe inside the SoC and observe all data transferred between the cores. Clearly, such an adversary would succeed, as the data passed between the PUF and the CC occurs in the open. However, this attack model is so extreme that only the most dedicated and motivated adversaries would undertake such a task. Similarly, users who are faced with such powerful adversaries are likely to have extensive resources themselves. Thus, these users are likely to shield the processor using known tamper-resistance techniques, and we find this threat to be minimal.

Moving away from the PPT model, we can return to the discussion of fault injection [163–167] and freezing [168] attacks. Fault injection attacks fail to threaten the confidentiality of the system, because these attacks are based on repeatedly inducing the fault with the same key. However, PUF ROKs can only be used once. At best, a fault injection would become a denial-of-service, as the key would not correctly enrypt or decrypt the message. Freezing attacks are similarly unsuccessful, because they operate on the assumption that the key existed in addressable memory at some point. However, that is not the case with PUF ROKs. These keys are generated dynamically and are never explicitly stored outside the processor itself. Thus, PUF ROKs offer robust defenses against these physical attacks.

One final class of attacks to consider is power analysis [176]. Simple power analysis (SPA) involves monitoring the system's power fluctuation to differentiate between portions of cryptographic algorithms. This information leakage can reveal how long, for instance, a modular exponentiation takes, which reveals information about the key itself. Differential power analysis (DPA) observes the power fluctuations over time by repeatedly executing the cryptographic algorithm *with the targeted key.* Ironically, DPA is considered harder to defend against than SPA. And yet, PUF ROKs are immune to DPA (since repeated execution is not allowed) while vulnerable to SPA. Even though SPA is a potential threat, known techniques can prevent these attacks [177].

7.6 Conclusion

In conclusion, this work presents a novel hardware-based approach to generating read-once keys (ROKs). Our underlying strategy is to integrate a PUF with a register to create a feedback loop. The result is that no data required for the PUF ROK ever exists outside of the processor itself. In addition, the feedback loop continuously overwrites the contents of the register, thereby destroying the key immediately upon use. As such, the design successfully captures the notion of a ROK.

In this work, we have defined a ROK in terms similar to a Turing machine. We presented our architectural design and proved that it matches the formalism. We described applications of PUF ROKs and addressed concerns regarding their practicality and usability. We presented details of our prototype design and shared insights regarding future production-quality implementations. We presented a security analysis of PUF ROKs under the PPT adversary model, and we also demonstrated that PUF ROKs are resilient against even more powerful adversaries with the ability to perform physical attacks on the device. In summary, we have demonstrated that PUF ROKs are both feasible and secure.

8 RESILIENT AUTHENTICATED EXECUTION OF CRITICAL APPLICATIONS IN UNTRUSTED ENVIRONMENTS

Up to this point, our focus has been on the challenges of defining contextual access control constraints and how to authenticate the necessary factors. In the next two chapters, we shift our focus to a more fundamental concern: the integrity of the execution environment. That is, we start with the observation that trusted enforcement of contextual policies depends on correct execution of the supporting application. In this chapter, we will explore how to provide execution integrity under the assumption that the OS kernel has become compromised.

8.1 The Necessity of Authenticated Execution

It is intuitive that execution integrity is necessary for proper enforcement of contextual access control. Specifically, corrupting the memory image of a trusted application can allow an adversary to bypass the security mechanisms. For instance, an adversary with control of the OS kernel could simply modify the value of a variable in memory, producing an unintended control flow sequence. Thus, policy models and authentication protocols are meaningless if the attacker can corrupt the application's execution environment.

A number of techniques exist for providing assurance of the application's integrity before execution. One can take a cryptographic hash of the program file on disk, and compare the result with a previously known value. Coupling this approach with attestation techniques can provide guarantees even for remote systems. These techniques have an inherent limitation, though, as they can only be used prior to executing the application. That is, once the program is running in memory, a corrupted OS or another vulnerable program could modify the program image, leading to unintended or catastrophic results.

In this chapter, we provide a novel approach for protecting the application's integrity while it is executing on a system. Our technique involves encoding the memory image into a protected space controlled by a trusted VMM. Previous attempts for providing such an integrity check have one major drawback: If corruption is detected, the application is terminated. In some instances for critical applications, termination is undesirable. Our work, in contrast, provides a recovery mechanism that allows the VMM to *repair* the memory image and continue running. In the setting of CDAC, this technique could allow a server to consider the integrity of a remote application before granting access to a trusted resource.

Several classes of applications, such as military, health or infrastructure monitoring software, are highly critical; compromising their correct execution may have dire consequences. Typically, such applications originate at trustworthy sources, and they undergo a thorough testing process, possibly including formal verification. Consequently, it is reasonable to consider these programs to be trusted and free of exploitable vulnerabilities. To ensure that the code executes correctly, it is vital to provide strong guarantees that the critical application is isolated from untrusted code. However, modern computing practices tend to make such isolation impossible.

Specifically, trusted software frequently runs on top of a commercial off-the-shelf (COTS) operating system (OS). Such COTS systems tend to be very complex and proprietary, preventing a rigorous security evaluation and formal analysis. Furthermore, the presence of legacy applications on the same machine may make secure alternatives infeasible. Proper isolation of the trusted processes, then, relies on the integrity and correctness of the OS. However, security vulnerabilities in the untrusted applications and/or the COTS OS destroy the guarantees of isolation and may lead to compromise of the trusted code.

Memory corruption attacks are among the most frequently occurring security violations, accounting for 70% of the total number of CERT advisories between 2000 and 2003 [178]. A common approach is for an attacker to provide malformed input to an application to change its execution. Such attacks can be especially devastating if the target is the OS itself. As today's attacks predominantly employ these tactics, the security literature has primarily focused on protecting an application from external sources and/or protecting the OS from compromise. However, recent high-profile attacks suggest that future adversaries may be very sophisticated, well funded, and state-backed, and may have very precise targets. For instance, Stuxnet combined multiple exploits in order to disrupt the proper execution of a particular programmable logic controller (PLC) [179]; the worm was harmless to machines that did not have this PLC installed.

To reflect this changing threat model, our work is based on a very powerful adversarial model, in which we assume that the OS has already been compromised. That is, the attacker has already "won the game," according to traditional security threat models. However, we assume that the adversary's goal is to leverage the OS privileges in order to modify the memory image of the critical application. For instance, if the critical application computes missile trajectories for a military operation, the adversary's goal may be simply to skew the results. Our goal, then, is twofold. First, we aim to identify such corruption whenever it occurs. Second, if the damage is reasonably small, we desire to *repair the memory image dynamically*, allowing the process to continue normal execution.

Application recovery is a complex task and may take place at different levels. Recovery-oriented computing (ROC) [87] involves designing rapid failure recovery mechanisms into new and existing applications. The resulting programs include the ability to "undo" errors by returning to a good state. The approach that we adopt in this paper is to use a trusted virtual machine monitor (VMM) to detect and repair the corrupted application memory pages. As such, our work can be seen as a technique for transparently incorporating ROC into the execution environment without modifying the original application code.

Existing research has already acknowledged the importance of protecting critical applications against an untrusted and/or compromised execution environment. Like ours, a common approach is to employ a trusted VMM that mediates interactions between the OS and the critical application, and restricts the kernel's access with respect to the memory space of the protected application. These VMM-based solutions generally fall into two broad categories: *memory authentication* and *memory duplication* approaches.

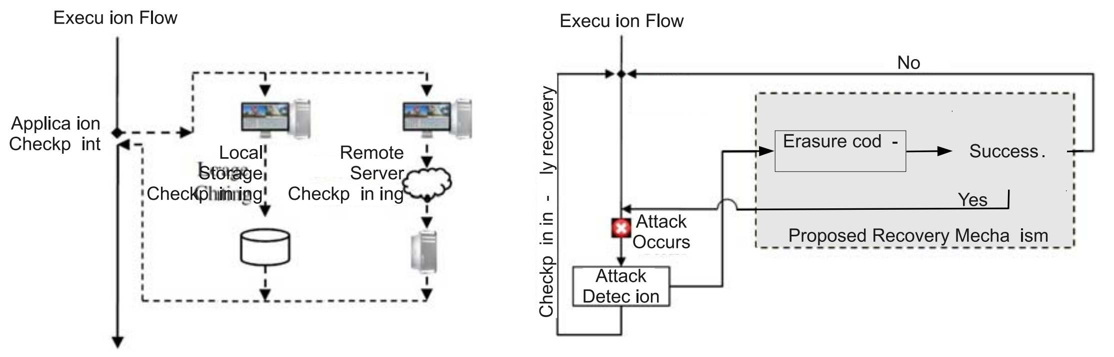
- *Memory Authentication.* In this category, a trusted component (*e.g.,* the VMM) applies cryptographic techniques to validate the integrity of the application's memory image during execution. For instance, Terra [67] provides trusted applications with isolated virtual machines, and uses cryptographic hashes of the software image to allow remote attestation to a third party. The hashes, along with a summary of the hardware and software layers running on the virtual machine, are signed with a private key stored in tamper-resistant hardware. Overshadow [84] provides both confidentiality and integrity of critical applications by encrypting the memory image. As the program executes, the VMM authenticates and decrypts pages as they are referenced.

- *Memory Duplication.* NICKLE [85] protects the OS from rootkits by securely storing a cloned image of the kernel. At boot time, the trusted VMM creates a copy of the kernel in a portion of memory that is inaccessible to the guest OS. As the system runs, only instructions retrieved from this copy are permitted to execute in kernel mode. That is, if a rootkit has been installed after the system boots, it *cannot* execute, as its instructions do not exist in the protected space. While NICKLE protects the OS at run-time, it cannot protect critical applications if the original kernel image is malicious.

Observe that these approaches primarily focus on *detection* of memory corruption. Memory authentication mechanisms terminate the protected application if corruption is detected. Similarly, memory duplication techniques detect and prevent the introduction of malicious *kernel* code, but do not protect application code. Furthermore, for critical applications, detection of corruption alone is inadequate, as disruption of the execution may have disastrous consequences. That is, the attacker still wins, even if the corrupted process is terminated.

To ensure critical application recovery, memory authentication must be combined with mechanisms for application checkpointing [180]. Such mechanisms allow the critical application to be re-instated to a valid configuration saved previously. Figure 8.1(a) illustrates alternatives for checkpointing: the memory image can be saved to protected local storage, or to a remote server if there are concerns that even the local storage may be compromised. However, checkpointing has two drawbacks. First, saving the memory image to an external system (either local or remote) incurs considerable overhead, therefore can only be done with limited frequency, otherwise most of the CPU time will have to be spent on checkpointing, rather than performing useful computation. Second, the *granularity* of recovery is coarse, as correct execution can only be resumed from the previously-saved checkpoint. However, between the time the last checkpoint was created and the attack is staged, the critical application performed more computation. The results of such computation are lost in the recovery process. The limited frequency of checkpointing (identified as the first drawback) only exacerbates the problem: the lower the frequency, the more significant the portions of the critical application execution that must be rolled-back.

We address these limitations through a novel framework for resilient execution of critical applications running in untrusted environments. Specifically, we propose and evaluate an *on-line recovery mechanism* that provides better performance and finer granularity compared to checkpointing. To meet these goals, we rely on a combination of memory image authentication and duplication. Figure 8.1(b) illustrates the functionality of the proposed approach: when corruption is detected, an on-line memory image reconstruction procedure is initiated, which aims at restoring the correct image based on a small amount of redundant information which is kept in protected


Execu ion Flow
Applica ion Checkp int
Local Storage Checkp in ing
Remote Server Checkp in ing
Execu ion Flow
No
Erasure cod -
Success.
Yes
Attack Occurs
Proposed Recovery Mecha ism
Attack Detec ion
Checkp in in - ly recovery

(a) Application Checkpointing

(b) On-line recovery does *not* require execution roll-back

Figure 8.1. Overview of proposed approach for on-line critical application recovery

storage, outside the reach of the OS or other untrusted software components. If reconstruction succeeds, then the application will continue with the exact same image that existed before the attack, without the need to roll back execution. Hence, the granularity of the recovery is optimal. If on-line reconstruction fails, then the system reverts back to the checkpointing procedure. However, as we show experimentally in Section 8.5, the success rate of on-line reconstruction is high under reasonable corruption attacks, resulting in a practical and efficient framework for achieving critical application availability.

For the memory duplication portion of our scheme, we have evaluated three approaches. First, the naïve approach simply clones the application memory image in its entirety. While efficient in terms of speed, the space requirements of this approach may be prohibitive. Second, we employ error correcting *digital fountain codes (DFC)* [181], which have been proposed for reliable communication over unreliable network channels. We focus on *LT* codes, a DFC instantiation, which take a message consisting of K blocks and encode the data into $N > K$ blocks. LT codes provide a probabilistic guarantee of reconstructing the original message if a small number of the N blocks are corrupted. Our LT approach is efficient in space, requiring less than 25% extra storage, but reconstruction must be done at every page reference. Our third approach, which provides a balance between the first two, is to use Reed-Solomon codes [88], which append an array of data with a small number of parity bits for error correction. While Reed-Solomon encoding and decoding is significantly slower than the same procedures for LT codes, these procedures must only be executed when corruption occurs, producing better aggregate performance.

8.2 Error Correcting Codes

Our system employs error correcting codes to recover corrupted memory images. We consider two codes: Reed-Solomon (RS) [88] which is a parity-checksum based code, as well as LT codes [89], a digital fountain rateless code.

RS codes rely on interpolation of polynomials in a finite field, and for each block of k symbols of input generates $n > k$ symbols of output. The resulting $RS(n, k)$ code can correct up to t symbol errors, where $t = (n - k)/2$. In our setting, each symbol is a byte of data. A common setting is $RS(255, 223)$ which can correct 16 bytes for each block of 223 bytes using a 32-bytes checksum. The checksum of a message, also referred to as a *syndrome*, is obtained by multiplying the k-byte message, interpreted as the coefficients of a k-degree polynomial, with $x^{(n-k)}$ and computing the remainder modulo a generator polynomial g of degree $2t$. Decoding is equivalent to computing the roots of a n-degree polynomial. The RS code is rather compute-intensive, but has the advantage that in absence of errors, the original message can be accessed directly, as opposed to fountain codes that are discussed next.

Digital Fountain Codes (DFC) have been designed for error correction in packet-switched communication networks, such as the Internet. In packet switched networks, data losses typically occur at the packet level, i.e., a packet is either correctly received, or it is entirely dropped (forwarding routers may decide to drop a packet along the transmission path if certain error-detection checksums fail). In this setting, having self-contained redundant information within each packet may not be an effective solution. The idea behind DFC is to re-encode the message to be sent, such that redundant error-correcting information is shared by multiple packets. Specifically, given K payload packets, DFC provide mechanisms that may generate a potentially infinite sequence of packets, and any $N > K$ such packets (where N is only slightly larger than K) are sufficient to re-construct the original message. Each transformed message is obtained by performing an exclusive-or operation on a sub-set of the original K packets. Due to their property that an infinite number of packets may be generated, such codes are called *rateless*. The term "fountain" is an analogy that captures the fact that the receiver needs to collect a number of *any* N packets to recover the message, similarly to collecting drops of water from under a fountain. We present the details of LT codes operation in the next section. The advantage of LT codes over RS codes is the faster encoding and decoding time. However, decoding has to be performed even if there are no errors, since the original message (i.e., memory block) is not available in direct form.

8.2.1 LT Codes

Figure 8.2 shows an example of block encoding in digital fountain codes (DFC). There are a total of three original message blocks ($K = 3$), which are encoded into four blocks ($N = 4$). Each original block M_i contributes to one or more of the encoded blocks. The functionality of the DFC is given by the mapping of original-to-encoded blocks, which can be specified as a bipartite graph G with $K+N$ nodes, corresponding to the K original blocks and the N encoded blocks. The graph G contains an edge (i, j), $1 \leq i \leq K$, $1 \leq j \leq N$, if the original block i contributed to the contents of encoded block j. With this notation, each encoded block can be expressed as

$$B_j = \bigoplus_{(i,j)\in G} M_i$$

The graph G could be chosen purely at random, e.g., for each encoded block j choose an in-degree d_j uniformly between 1 and K, and then set B_j as the exclusive-or of d_j blocks M_i chosen uniformly at random. However, as shown in [181], even if the probability of successful reconstruction of such a scheme may be good for N slightly larger than K, the average in-degree of encoded blocks is $K/2$, which determines high encoding cost. We employ an efficient DFC instance called *LT codes* [89]. LT codes use a sparse graph for which the in-degree of an encoded block is chosen at random from a *robust soliton* distribution. As a result, the average degree $\bar{d}$ of the graph G is much less than $K/2$. Next, we detail how message encoding and decoding is performed.

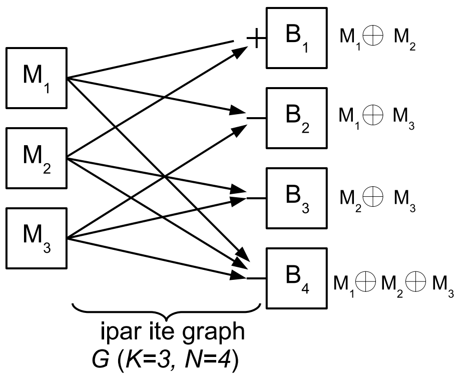


Figure 8.2. Digital Fountain Codes: Block Encoding

8.2.2 Block Encoding

The main operation of encoding is finding the in-degree d_j for each encoded block B_j. Once d_j is determined, the actual source blocks are randomly chosen from the original K message blocks. To find d_j, we first define the *ideal soliton distribution*, which has the form [181]:

$$\rho(d) = \begin{cases} \frac{1}{K} & , d = 1 \\ \frac{1}{d \cdot (d-1)} & , otherwise \end{cases}$$

Let $0 < c < 1$ be an arbitrary constant, and denote by δ the desired probability for successfully recovering blocks. Define

$$S = c \cdot \sqrt{K} \cdot \ln \frac{K}{\delta}$$

and let

$$\tau(d) = \begin{cases} \frac{S}{K} \cdot \frac{1}{d} & , d = 1, 2, \ldots, \frac{K}{S} - 1 \\ \frac{S}{K} \cdot \log \frac{S}{\delta} & , d = \frac{K}{S} \\ 0 & , d > \frac{K}{S} \end{cases}$$

Then, the *robust* soliton distribution [181] of degree d is given by

$$\mu(d) = \frac{\rho(d) + \tau(d)}{\sum_d (\rho(d) + \mu(d))}$$

where the denominator is a normalizing factor that ensures that the probability is bounded in the interval $[0, 1]$. Algorithm 7 shows the pseudocode for encoding the contents of an application's memory image.

Algorithm 7: EncodeMemoryImage

```
Input: $M_1, \ldots, M_K$ : original memory blocks ; $N$ : the number of encoded (i.e.,
       output) memory blocks
Output: $B_1, \ldots, B_N$ : the encoded blocks

for $j := 1$ to $N$ do
    /* choose in-degree for encoded block $j$ */ ;
    choose $d_j$ randomly from distribution $\mu(d)$;
    $B \leftarrow 0$ ;
    for $i := 1$ to $d_j$ do
        choose $M$ uniformly at random from $M_1, \ldots, M_K$;
        $B \leftarrow B \oplus M$;
    $B_j \leftarrow B$ ;
```

8.2.3 Block Decoding

The algorithm for decoding inspects the graph G to find an encoded block that has in-degree equal to 1. Due to the fact that in-degrees are drawn from a robust soliton distribution, such a block exists with high probability. Denote such a block by B'. It immediately results that there exists i with $1 \leq i \leq K$ such that $M_i = B'$. Next, the algorithm performs an exclusive-or between B' and all encoded blocks that have M_i as one of their input blocks, i.e., all B_j such that $(i, j) \in G$. The in-degree of all blocks B_j is decreased by 1.

The algorithm then continues recursively on the modified set of encoded blocks, by finding another block B'' with in-degree 1. The process continues until either *(i)* all blocks have been recovered, or *(ii)* no encoded block of degree 1 can be found. In the former case, the decoding is successful, whereas the latter case signifies that decoding failed. However, due to the distribution of block degrees, the probability of successful decoding is high. In practice, the success probability of the decoding is $1 - \delta$, where δ is a small positive constant depending on K and N [181].

Algorithm 8 shows the pseudocode for decoding the contents of an application's memory image.

Algorithm 8: DecodeMemoryImage

```
Input: $B_1, \ldots, B_N$ : encoded memory blocks ; $K$ : the number of original blocks to
       be recovered
Output: $M_1, \ldots, M_K$ : the decoded blocks
for $i := 1$ to $K$ do
    if ( $\nexists\, j$ such that $in\_degree(B_j) = 1$) then
        abort with error ;
    else
        find the unique $s$ such that $(s, j) \in G$ ;
        $M_s \leftarrow B_j$ ;
        foreach $k, 1 \leq k \leq N$ such that $(s, k) \in G$ do
            $B_k \leftarrow B_k \oplus M_s$ ;
```

8.3 System Overview

Our system design consists of using a VMM to incorporate error-correcting codes into the x86 memory architecture. The goal of our design is to protect the code and data for applications executing within an untrusted OS environment. That is, we aim to provide a robust defensive mechanism that protects the application even if the OS kernel has been corrupted. As described in Section 2.5, existing work in this area that has been designed to protect application data has provided only a detect-and-terminate approach. Our goal is more ambitious, as we want to provide a probabilistic guarantee that the application can recover from memory corruption and continue processing.

8.3.1 Architecture and Approach

Figure 8.3 shows a high-level view of our system. The guest OS executes within an *untrusted zone* (indicated in red) that consists of an emulated memory that is established by the trusted VMM. The guest OS has complete control over the emulated memory, as well as the corresponding swap file. Our goal is to execute trusted guest applications within this untrusted zone by leveraging the VMM.

Before the guest OS boots, the VMM establishes a segment of protected physical memory that is beyond the reach of the guest OS. When a guest application runs, the VMM allocates a portion of this memory to store verification and reconstruction data (*e.g.,* hashes, redundant blocks) that is specific to that application. As the system runs, it alternates between *user mode*, in which the application executes, and *kernel mode*, in which the guest OS has control of the system.

Our basic approach is to use the VMM to detect these mode switches and perform some additional work. When the switch is from user mode to kernel mode, we take a snapshot of the application and encode the application to generate some redundant data. Later, when the mode switches back from kernel mode to user mode, we check the integrity of the application's memory image and, if necessary, use the redundant data to repair any corruption that has occurred.

As a part of the encoding process, the VMM generates a small amount of randomized initialization data before the guest OS boots. As this data is inaccessible to the guest OS or applications, and is randomized at run-time, it thwarts static attack strategies. Also, while our proof-of-concept implementation uses a VMM that executes within a host OS, our architecture also applies to cases when there is no host OS, and the VMM executes on bare hardware. This is how Xen, for example, operates. The advantage of eliminating the host OS is that it reduces the trusted code base.

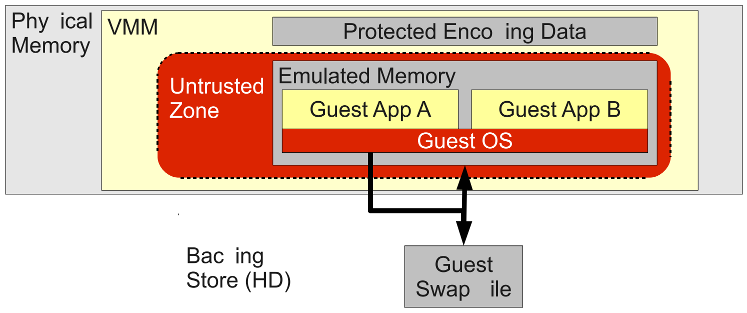


Figure 8.3. VMM-based System Architecture

An important aspect of Figure 8.3 is the delineation of the trust barrier. As the untrusted guest OS has full access to the emulated memory space, we make the explicit assumption that everything inside the dotted line is untrusted. Furthermore, the guest OS has access to external devices, such as a hard drive. This access has a direct implication on our system design, as the guest OS may swap a page of emulated memory to the hard drive in response to a *page fault*. However, as we will explain in the next section, the VMM can detect the presence of such an interrupt and monitor the corresponding write. In this way, the VMM can continue to monitor the contents of memory, even if they get swapped out to disk. The arrows in the figure indicate that it is the guest OS that controls which emulated pages of memory are swapped out.

8.3.2 Attack Model

The aim of our system is to provide a resilient execution environment for *trusted* guest applications. That is, we assume the protected application has undergone extensive analysis, possibly including formal methods, to verify that the code is immune from common vulnerabilities, such as buffer overflows. As such, we do not consider any attacks exploiting the application itself. However, the rest of the virtual environment, including the guest OS and other guest applications, is generally untrusted. In Section 8.4, we describe minor exceptions to this assumption. We consider the underlying host OS (if there is one, which is not true for VMMs like Xen) and the VMM to be trusted (assumptions that are common in the literature on virtualization-based security).

As a description of a sample attack, consider a trusted application running on a guest platform that also has a vulnerable network-facing application, such as an email client. Our model is that an attacker exploits a vulnerability in the untrusted application to inject code in the OS. The injected code then corrupts that application's page tables to point to the trusted application, and then modifies the trusted application's memory contents. Thus, the memory corruption originates from a source outside of the trusted application. We also assume that the attacker corrupts *only a small portion of the trusted application.* This is consistent with scenarios where attackers aim to remain stealthy, while causing the application to deviate from the correct execution flow.

8.4 System Details

Our goal is to recover from memory corruption attacks by applying error-correcting codes to the contents of memory. When an application is running in user mode, the

CPU has a Current Privilege Level (CPL) of 3; when the OS kernel takes over, the CPL switches to 0. Inside the VMM, we detect whenever the CPL switches from user mode to kernel mode (*i.e.,* the CPL switches from 3 to 0). When this occurs, we take, in essence, a snapshot of the current application memory image. If the kernel is malicious and then corrupts the contents of memory, we attempt to repair the damage by restoring the snapshot when the application continues processing. Before describing the details of how our solution works, we provide a small amount of relevant background material describing the x86 memory layout.

8.4.1 x86 Memory Layout

Figure 8.4 illustrates the key components of the x86 memory layout and virtual memory addressing techniques for two applications[1]. The virtual memory space of each application is divided into a sequence of pages, typically 4KB each. When an application references a virtual memory address, the virtual page must be translated into a *frame* in physical memory. This translation process is handled by the hardware *memory mapping unit* (MMU).

The address of the frame is primarily found using two techniques. First, a *translation lookaside buffer* (TLB), which resides in high-speed cache, may contain the frame address if the page has been recently accessed. If the TLB does not contain the address, then a *page walk* is initiated. In a page walk, the control register CR3 is used to locate the page directory (PD in the figure), which is an array of entries for locating page tables (PT in the figure). Depending on the page size, one or more page tables will be traversed. Finally, the physical address for the frame will be found as an entry in the page table. For increased performance, any frame, including the page directory and page tables, may be stored in a high-speed cache.

[1]Technically, we should make a distinction between the *application*, which is a user-level abstraction, and the *process*, which is a unique OS-level thread of execution. Moreover, an application may consist of multiple processes. However, since our discussion centers on the idea of a *trusted application*, we will primarily use that term.

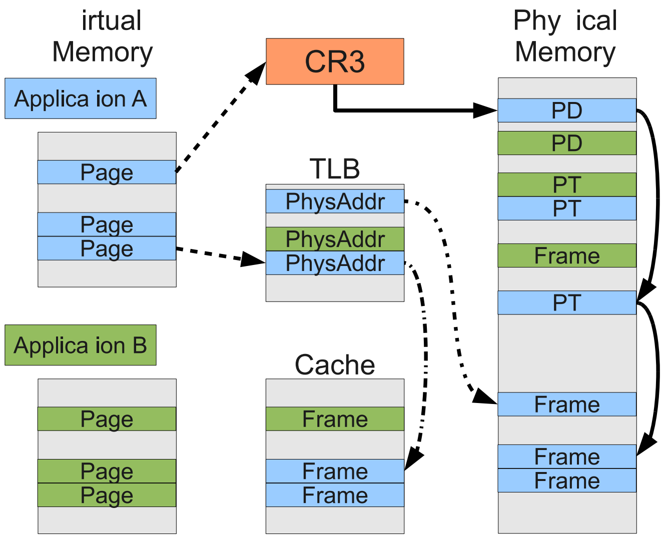


Figure 8.4. Overview of the x86 memory layout, including page tables and page directories

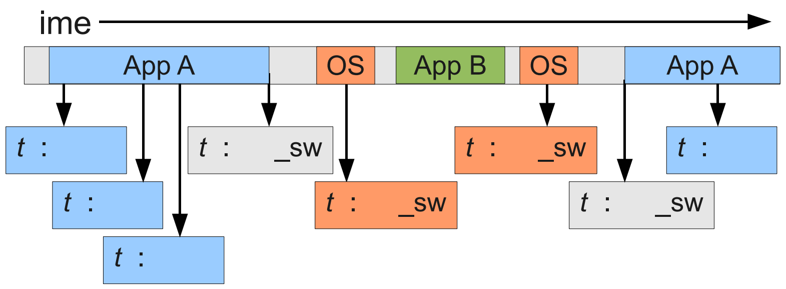


Figure 8.5. Key events as execution progresses. Unlabeled time in gray indicates VMM execution.

Figure 8.4 does not show how *demand paging* influences memory, though. In demand paging, if a page has not been previously accessed, then there is no corresponding physical frame. Rather, when the first access occurs, a *page fault* causes the OS kernel to load the data from a *backing store*, typically a hard drive. Once the frame is loaded and the page tables have been updated, the application continues executing as expected.

In our solution, we have integrated LT and Reed-Solomon codes into the physical memory access of the VMM, and we have accomplished this using a technique similar to demand paging. Specifically, we decode a frame corresponding to user-mode applications only when the application actually references that page. When control of the CPU is taken away from the application, we then encode all referenced frames and store checksums to ensure the integrity of each frame. We also store redundant information to help recovery. After the encoding, the OS kernel executes. If the kernel attempts to corrupt the application in any way, our decoding techniques can detect the tampering. Furthermore, we provide probabilistic guarantees that our techniques can *repair* the memory image even if corruption has occurred. Note that our techniques are operating directly on the frames themselves, regardless of whether they are stored in main memory or in a cache.

8.4.2 Memory Encoding and Decoding

Figure 8.5 shows a sampling of some key events during execution of our VMM-based protection mechanism. In a multitasking system, user mode applications execute for small periods of time, called *quantums*. The quantum ends when a hardware interrupt occurs or the application issues a *system call*, which is a request for a service from the OS kernel. When this occurs, the CPU performs a *mode switch* from user mode (CPL 3) to kernel mode (CPL 0). As this switch occurs in hardware, the VMM is able to detect the switch and interrupt processing. Another mode switch also occurs when returning to user mode. The events at times t_4 and t_7, denoted *md_sw*, correspond to mode switches. When the OS kernel itself is executing, it may issue a *context switch* (*ctx_sw*), as indicated at times t_5 and t_6. In a context switch, the kernel performs a number of tasks, including updating the CR3 register for a different application. The gray time periods in Figure 8.5 correspond to the execution of the VMM.

The influence of demand paging on our design becomes clear when considering the events at times t_1, t_2, t_3, and t_8. When a quantum begins (at a mode switch from CPL 0 to 3), the application's memory image is in an encoded form. However, we only decode a frame *when it is first accessed*. So, at time t_1, application A performs a write to page p_1 (which we assume is the first reference to p_1 for illustration). When the frame is located, the VMM interrupts processing to decode the frame. Then, at time t_2, the VMM does not have to do any work, as p_1 is already decoded. At time t_3, p_2 must be decoded.

When the mode switch occurs at time t_4, the VMM must encode all pages that have been decoded during the previous quantum. In this figure, that means p_1 and p_2 must be encoded. The system will then continue processing as usual until the mode switch at t_7. Just like before, we do nothing immediately. However, at t_8, the frame for page p_2 must again be decoded, because it has not previously been accessed during this quantum.

The advantages of our approach are twofold. First, by performing the decoding on-demand, we significantly reduce the overhead our system imposes, as we will show experimentally in Section 8.5. Second, by performing the encoding of frames at the mode switches, we are ensuring that *no kernel instruction will execute before encoding occurs*. Thus, if the kernel has become corrupted, the VMM successfully protects the application from attack. We implemented two versions of our design, with LT codes and Reed-Solomon codes respectively. We provide the details next.

8.4.3 LT Codes-based Approach

Algorithm 9 describes the encoding process using LT codes. Each frame is partitioned (routine *Partition*) by interpreting it as a sequential array of K blocks of equal size. These blocks are encoded (routine *Encode*) into $N > K$ blocks according to a bipartite graph (see Appendix 8.2.1). The first K encoded blocks are copied back into the frame in memory, whereas the remaining $N - K$ are stored in the VMM protected space. The VMM also stores the hash of each of the first K blocks, as well as the hash of the original decoded frame.

The VMM stores a global array of data structures, denoted as *StoredData*. Each data structure stores the extra $N-K$ blocks plus the hashes of the K blocks stored in OS-accessible memory. The structure *EncodingData* consists of *EncodingData.blocks* (the array of $N-K$ extra blocks), *EncodingData.blockhash* (the array of hashes for the K blocks in memory), and *EncodingData.hash* (the hash of the entire frame). Its counterpart *DecodingData* is used for decoding in Algorithm 10 (the *Decode* routine is presented in Appendix 8.2.1).

Algorithm 9: EncodeMemoryImageWithLT

```
Input: $ReferencedFrames$: an array containing the frame numbers referenced
       during the quantum
$i \leftarrow 0$ ;
while ($i < ReferencedFrames.length$) do
    $FrameNumber \leftarrow ReferencedFrames[i]$;
    $Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
    $EncodingData.hash \leftarrow$ Hash($Frame$) ;
    $KBlocks \leftarrow$ Partition($Frame$);
    $NBlocks \leftarrow$ Encode($KBlocks$);
    for $j := 0$ to $K-1$ do
        $EncodingData.blockhash[j] \leftarrow$ Hash($NBlocks[j]$);
        write $NBlocks[j]$ to $Frame$ ;
    for $j := K$ to $N-1$ do
        $EncodingData.blocks[j-K] \leftarrow NBlocks[j]$;
    $StoredData[FrameNumber] \leftarrow EncodingData$ ;
    $i \leftarrow i+1$ ;
```

8.4.4 Reed-Solomon Codes-based Approach

The other implementation of our approach was to use Reed-Solomon codes. The disadvantage of Reed-Solomon codes is that the encoding and decoding process takes longer (see Section 8.5). However, Reed-Solomon allows us to keep the contents of main memory intact. Specifically, a Reed-Solomon code takes an array of k symbols, and appends it with $n-k$ symbols, called the *syndrome*. At decoding time, Reed-

Algorithm 10: DecodeSingleReferencedPageWithLT

```
Input: $FrameNumber$: the index of the frame to be decoded
$Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
$NBlocks \leftarrow$ Partition($Frame$);
$DecodingData \leftarrow StoredData[FrameNumber]$;
$DecodeBlocks \leftarrow \emptyset$;
for $i := 0$ to $K - 1$ do
    if $Hash(NBlocks[i]) = DecodingData.blockhash[i]$ then
        append $NBlocks[i]$ to $DecodeBlocks$ ;
for $i := 0$ to $N - K - 1$ do
    append $DecodingData.blocks[i]$ to $DecodeBlocks$;
$KBlocks \leftarrow$ Decode($DecodeBlocks$);
write $KBlocks$ to $Frame$;
if $DecodingData.hash = Hash(Frame)$ then
    abort with error;
```

Solomon can then correct up to $(n - k)/2$ corrupted symbols to recover the original data. As such, we only need to store the syndrome, and we do not need to write anything back to main memory.

The fact that main memory is not modified during the encoding process allows for an optimization: If the page was only referenced for read operations (*i.e.,* it has not been modified), then we do not have to perform the encoding of that frame. Furthermore, we only need to store a single hash for the entire frame. At decoding time, if the hash of the frame matches the stored value, there is no need to decode. These optimizations make the Reed-Solomon approach very advantageous when most memory accesses are simply read operations.

There is one other important consideration for the Reed-Solomon implementation that did not arise in the LT code version. Specifically, we do not want to partition the page simply into an array of arrays. If we were to do so, the adversary would only need to corrupt $\lceil (n - k)/2 \rceil + 1$ consecutive bytes to successfully corrupt the frame. That is, when the Reed-Solomon parameters are $(255, 223)$, corrupting 17 bytes would be sufficient for a successful attack (see Section 8.2).

To mitigate this threat, the VMM generates a random mapping for bytes within a frame at run-time. For a 4KB frame, each byte is assigned a random value from 0 up to $\lceil 4096/k \rceil - 1$, where this value indicates one of the arrays. So, if $(255, 223)$ codes are used, there are $\lceil 4096/223 \rceil = 19$ arrays, and each byte would be assigned to one of the 19 arrays. However, the randomization ensures that the first 18 arrays will each have 223 bytes assigned to them. The last array will get the remaining 82 bytes, and will be padded with 0 to also have length 223. Hence, when encoding or decoding, the *Partition* and *Departition* functions loop through the 4096 bytes of the frame, copying each byte to the corresponding array. Figure 8.6 illustrates the randomized mapping for the $(255, 223)$ encoding. As we will show later in Section 8.5, this randomization reduces the attack success rate. The attacker must now corrupt 305 bytes under the $(255, 223)$ parameters to guarantee success. Specifically, there are 19 arrays, and each can tolerate 16 corrupt symbols; 305 corrupt bytes guarantees that at least one array has 17 corrupt symbols.

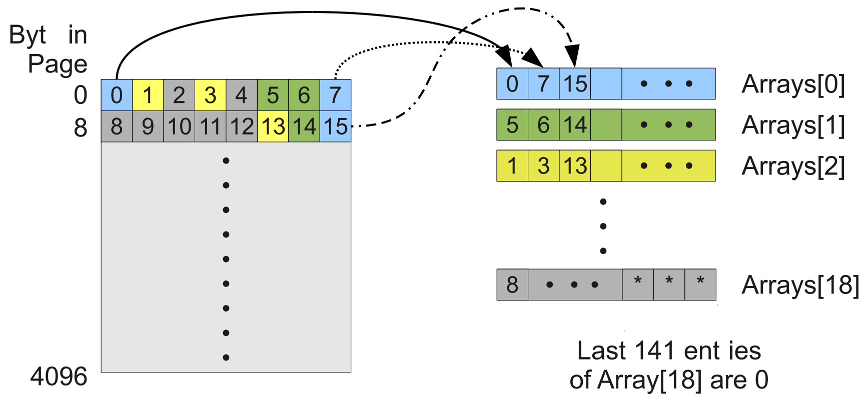


Figure 8.6. Randomized mapping of bytes in a page to 19 arrays for Reed-Solomon (255,223) encoding. Each Arrays[i] has 223 entries. Since Arrays[18] only receives 82 bytes from the page, that array is padded with values of 0 to get a length of 223 bytes.

Algorithm 11 shows the algorithm for encoding the referenced frames under the Reed-Solomon approach. As before, *StoredData* is an array of data structures stored in the protected VMM space. In this case, *StoredData* consists of a single SHA-1 value *StoredData.hash*, an array of syndromes *StoredData.syndromes*, and a flag to indicate the page has been modified[2]. When the mode switch occurs, the VMM checks to see if the page has been modified by checking the flag. If the page has not been modified, there is no encoding that is needed. If the flag indicates a write has occurred, we hash the page to see if the write actually changed the contents (*i.e.,* it did not write a value that matched what was already stored). Only if the page has actually modified do we partition the frame and create the syndromes.

Algorithm 11: EncodeMemoryImageWithRS

```
Input: $ReferencedFrames$: an array containing the frame numbers referenced
       during the quantum
$i \leftarrow 0$ ;
while ($i < ReferencedFrames.length$) do
    $FrameNumber \leftarrow ReferencedFrames[i]$
    $Frame \leftarrow$ GetPhysicalFrame($FrameNumber$)
    $EncodingData \leftarrow StoredData[FrameNumber]$
    if $EncodingData.flag = 0$ then
        return ;
    $h \leftarrow$ Hash($Frame$);
    if $EncodingData.hash = h$ then
        return ;
    $EncodingData.hash \leftarrow h$;
    $Arrays \leftarrow$ Partition($Frame$);
    pad $Arrays[Arrays.length - 1]$ with 0 for correct length ;
    for $j := 0$ to $Arrays.length$ do
        $EncodingData.syndrome \leftarrow$ Encode($Arrays[j]$);
    $i \leftarrow i + 1$ ;
```

Algorithm 12 describes the algorithm for decoding the frame under the Reed-Solomon approach. Note the optimization in lines 3-5: If the page's hash matches the stored hash value, then there is no need to do any decoding.

[2]We could actually use the dirty bit in the corresponding page table entry for this purpose, but that would require doing a page walk. Hence, we chose to sacrifice a small amount of space to optimize for speed.

Algorithm 12: DecodeSingleReferencedPageWithRS

```
Input: $FrameNumber$: the index of the frame to be decoded
$Frame \leftarrow$ GetPhysicalFrame($FrameNumber$);
$DecodingData \leftarrow StoredData[FrameNumber]$;
if $DecodingData.hash$ = $Hash(Frame)$ then
    return ;
$Arrays \leftarrow$ Partition($Frame$);
pad $Arrays[Arrays.length - 1]$ with 0 for correct length ;
for $i := 0$ to $Arrays.length$ do
    append $DecodingData.syndrome[i]$ to $Arrays[i]$;
    $Decoded \leftarrow$ Decode($Arrays[i]$);
    Departition($Decoded$);
if $DecodingData.hash$ = $Hash(Frame)$ then
    abort with error ;
```

8.4.5 Storage of Recovery Information

Figure 9.1(d) shows the structure of data storage for a single page under both schemes. In the LT approach, we assume $K = 16$ and $N = 18$. The column on the left shows the original contents of memory. When the data is encoded, the contents of physical memory are then overwritten with the encoded blocks. The additional two blocks are also stored in the VMM, along with the hashes of the blocks stored in physical memory. A hash of the original (unencoded) data is stored for an integrity check. In this figure, we do not show the encoding graph[3] (120 bytes), as all pages share the same graph.

In the Reed-Solomon scheme, we assume the parameters are $(255, 223)$. To generate the encoding data, the original frame (on the left) is partitioned into 19 arrays of 223 bytes each. The last array consists of 82 bytes from the frame, followed by a padding of 141 bytes with the value 0. For each array, we generate and store a 32-byte syndrome. The one-byte flag is used to indicate that the page has been modified, and the hash of the original (unencoded) data is stored for an integrity check. In this figure, we do not show the page-to-array mapping, which is simply an array of 4096 bytes.

[3]For details about the encoding graph concept, please see Appendix 8.2.1.

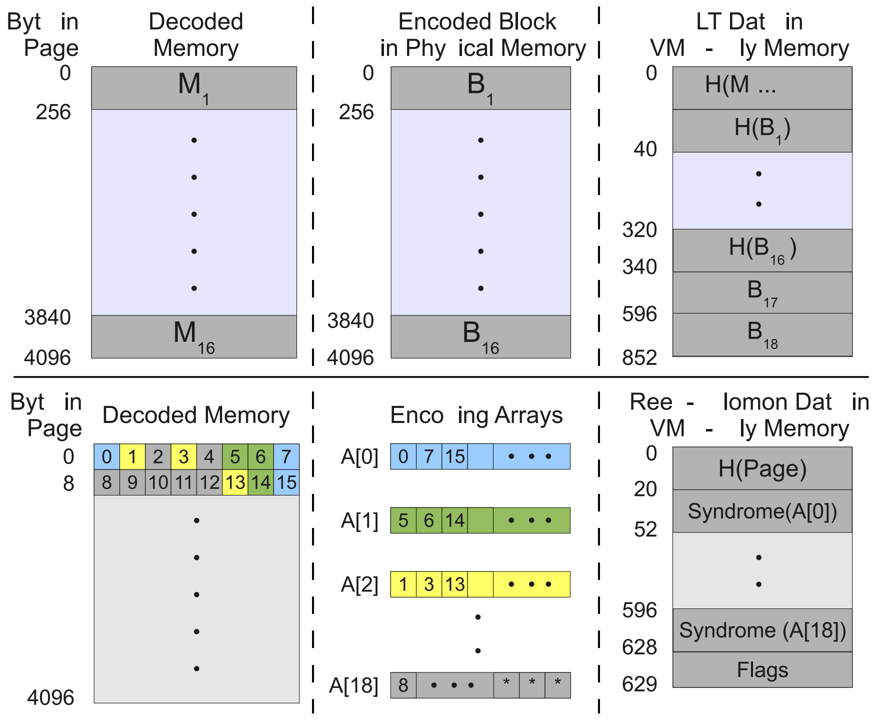


Figure 8.7. Decoded memory, intermediate and stored data for LT (top) and Reed-Solomon (bottom).

8.4.6 Implementation Challenges

Integrating memory encoding into a real system requires addressing a number of issues. First, the VMM must become aware of the OS paging procedure. That is, when the guest OS kernel responds to a page fault, it may choose to swap an application page to disk. Once the data is on the disk, it is beyond the protection of our memory encoding scheme and subject to attack. However, as page faults trigger a hardware interrupt, the VMM can detect this occurrence and monitor for the OS initiating the transfer of a page to the backing store. Later, when that same data is swapped back into memory (as the result of another page fault), the VMM can update its data structures to map the encoding data to the new frame. If the data was corrupted while on disk, the hash will not match when the frame is later decoded.

The next concern is the preservation of state values contained in CPU registers. Specifically, when a mode switch occurs, the registers are storing the current values of some application variables. If the kernel initiates a context switch, it pushes those values onto the application's stack for preservation. When another context switch occurs to return to the original application, these values are popped off the stack and restored to the registers. If the OS is corrupt, it could modify the values stored in the registers before the mode switch returns control to the application. To prevent this, the VMM, in essence, duplicates the context switch work. We use the value stored in the CR3, which is unique to each application[4], to locate a storage place for the state data.

The last, and most critical challenge, is the distinction between legitimate page modifications and possible attacks from the kernel. Specifically, the main purpose of the OS kernel is to provide user-mode applications with access to hardware resources, such as the network card or hard drive. This access is granted through the *system call* interface. In order for the application to receive the requested data, the kernel must be able to write to a memory location in the application's memory space, including the stack. That means that, when the mode switch returns control to the application, the hash of those frames will not be correct.

To accommodate this, two techniques are required. First, the frame storing the top of the stack will not be encoded. Rather, the VMM will monitor that frame to ensure that the only changes made are appropriate for responding to a system call (*e.g.,* the memory storing the index of the desired system call will be popped by the kernel). Second, we must reserve a sequence of pages in the virtual address space that will not be encoded. The kernel can write data to these pages, then the application can copy the data to an appropriate location, such as its heap. These two techniques allow the kernel to perform legitimate memory writes, while preventing full access to the memory space.

[4]In theory, the CR3 could change if the application's page directory is swapped out. This happens rarely in practice, though. To be thorough, the VMM, which is aware of the page directories, can track this and react accordingly.

8.4.7 Threat Analysis

Our stated goal is to protect trusted guest applications from memory corruption attacks launched by the guest OS or other guest applications. We provide this protection through the combination of the encoding scheme and verifying the integrity of encoded blocks with a cryptographic hash function. While our design offers a robust layer of protection, we have identified a number of remaining threats. First, our system must trust the software that exists outside the trust boundary (Figure 8.3). Specifically, the VMM must be considered trusted. Also, if a host OS exists below the VMM, it must also be trusted. This also means that if the host OS swaps a frame out to disk, an attacker with full access to the disk could potentially attack the system. Thus, eliminating the host OS (*i.e.,* using a VMM that runs on bare hardware) helps to reduce the assumption of trust.

Corruption attacks. In order for an attacker to corrupt a guest application successfully, he would first have to construct a collision under the hash algorithm. To complicate that search, the collision must be the same size as the encoded block (in the LT approach) or the full page (in the Reed-Solomon approach). If, at any point, the required hash does not match, that portion of memory may be discarded. Furthermore, if the final page does not hash correctly (*i.e.,* our decoding failed), the system will simply abort. While the use of SHA-1 makes finding a collision difficult (that is, nearly impossible in practice), the run-time randomization of the LT encoding graph and the Reed-Solomon partition map make the attack even more difficult. As these data structures are inaccessible to the guest OS and applications, the attacker would have to resort to guessing. Thus, we find successful corruption attacks from other guest applications to be extremely implausible, though possible in theory.

Denial of service. While our system assumes that the guest OS may be compromised, we are assuming that it is still somewhat functional. That is, if the attackers goal is merely to shut down the system, she could corrupt the OS beyond repair. This attack is outside our scope, as we focus on scenarios where the attacker's goal is to keep the system working in a compromised fashion. A more subtle attack could aim to prevent the OS from scheduling the trusted application. To defend against this attack, we suggest deploying a technique similar to a watchdog timer. That is, the application may send a report to an external monitor at least once within a certain time period. If the application has not been running, the fact that the monitor does not receive the report will indicate that this attack has occurred. At that point, external administration would be required to restart or repair the system. To prevent the guest OS from forging such reports, cryptographic techniques can be applied to network data, as described below.

VMM exploits. A software VMM is subject to vulnerabilities just like any other piece of code. Obviously, if the adversary can corrupt the VMM, then our system offers no guarantees of security. However, a number of projects have made great strides toward protecting the VMM from corruption [65, 182, 183]. As such, we simply rely on an assumption of trust in the VMM.

Application vulnerabilities. Our design assumes that the protected application is vulnerability-free. Clearly, such an assumption cannot be made in general, as buffer overflows and other vulnerabilities remain a common problem. However, many environments require trusted applications that have undergone formal analysis and verification. Our design offers these applications a resilient execution environment that protects the application from *external* threats.

Corrupted network data. If the critical application sends or receives data across a network interface, the guest OS kernel has an opportunity to modify this data in transit. Specifically, access to the network card is set up by the OS via a system call. In the case of verifying received data, a public key could be hard-coded into the application without incurring a significant threat. Securing private or symmetric keys, though, is not possible in our current approach, as the guest OS has full access to read that application's memory space and would be able to forge messages easily. Instead, for applications that require secret keys, we propose combining our approach on top of existing solutions, such as Overshadow [84][5] or a virtual Trusted Platform Module [59]. By utilizing other protection mechanisms, the application could have secured access to a secret key as needed.

8.4.8 Fail-safe Recovery

Recall that our goal was to detect corruption in all cases and to provide a mechanism to repair the application dynamically without interrupting service. Clearly, our use of cryptographic hash functions to authenticate the integrity of each memory page before it is accessed accomplishes the former[6]. We have also proposed a scheme that provides a *probabilistic guarantee* of the latter. That is, our system, as described above, may not be able to recover transparently, especially if the amount of corruption within a single page is large.

Observe that, if the recovery fails, the VMM is immediately aware of the failure, and knows exactly which page in the application's linear address space is corrupted. As the trusted component, the VMM would have full access to replace the memory image as necessary. If the corrupted page consists of code or read-only data, the VMM can extract the relevant portion of the application from a protected local storage that is inaccessible to the guest OS. This procedure would allow the application to continue processing as before.

The more problematic failure is if the corrupted page contains data that has been modified at run-time. Clearly, the VMM has insufficient information to initiate a transparent local recovery. Instead, the VMM would initiate a remote recovery protocol with a trusted server. Note that this communication channel would not be threatened by the guest OS, as the VMM would have direct control of the network interface. The VMM would then re-instate the memory image saved at the last

[5]Recall that Overshadow ensures confidentiality and integrity by encrypting the memory image. However, our goals are integrity and *availability*, the latter of which is not addressed by Overshadow. As such, we see our schemes as complementary, and believe they could be combined for even greater protection.

[6]As of this writing, collision attacks in SHA-1 are not practical. If this threat is a concern, one could upgrade the hash function to SHA-256 or other stronger functions.

checkpoint, as shown in Figure 8.1(a). Note that, in that case part of the execution must be rolled back.

8.5 Implementation & Evaluation

To provide a proof-of-concept, we have implemented a preliminary prototype to measure the performance impact of our system design. We have modified the source code for version 0.9.0 of the QEMU hardware emulator [184]. We adapted the LT code library developed by Uyeda *et al.* [185], as well as the Reed-Solomon implementation by Rockliff[7]. Whenever an operation triggers a mode switch (*i.e.,* the CPL switches from 3 to 0), we make a call to a custom procedure that encodes the recently referenced pages as described in Section 8.4. Our implementation also detects when a page is first referenced during a quantum. When that occurs, we perform the decoding algorithm as described above.

8.5.1 Performance Overhead

Our test platform consisted of a 2.26GHz Intel® Core$^{\text{TM}}$ 2 Duo CPU with 3GB of 667MHz memory, running Ubuntu 10.04 (Lucid Lynx), with version 2.6.32-29 of the Linux kernel as the host OS. Our VMM was an adaptation of QEMU version 0.9.0, and the guest virtual machine was running Redhat 9 with version 2.6.20 of the Linux kernel on 128MB of emulated physical memory.

Our baseline result is the naïve implementation, in which the entire memory image is cloned in the protected space. For the baseline implementation, we could have simply copied the memory image from the clone without any integrity check. However, we opted to perform the SHA-1 calculation in the baseline. Doing so allows us to identify how much overhead occurs as a result of the hash, and how much occurs as a result of the encoding/decoding schemes.

[7]Rockliff's Reed-Solomon code software can be downloaded at `http://www.eccpage.com/`

For both the baseline and the Reed-Solomon cases, we strove to ensure that we measured the difference that would result between doing a full frame encoding/decoding as compared to one that did not require encoding/decoding. Recall that if the page has not been modified (*i.e.,* the current hash matches the previous), then no additional work is required. To measure the difference, we used a ratio that ensured approximately 1% of all pages referenced required the full processing. The remaining 99% required only the initial SHA-1 hash calculation.

Figure 8.1 summarizes the performance impact for the *nbench* benchmark suite [186]. Clearly, incorporating any additional work into the mode switch significantly increases the overhead relative to a normal mode switch. This is expected, as a normal mode switch only requires about 3.2μs (as illustrated in the "without encoding" line of the table). However, when the total time for encoding and decoding is considered relative to the amount of time spent executing user mode applications (which significantly dominates the total system processing time), the performance impact of our system is quite reasonable. Our LT implementation imposes approximately an 10.4% performance overhead, relative to the total computational time. The Reed-Solomon version entails a more reasonable 3.8% overhead. Observe that most of the Reed-Solomon aggregate overhead is a result of the SHA-1 integrity calculations. This can be seen in the baseline, which incurs a 3.5% overhead just doing the hash and copying the page with no encoding.

One should also observe that, while the Reed-Solomon impact is lower than the LT version, the better performance is a direct result of the fact that we only need to perform the Reed-Solomon computation on rare occasions. If an encoding or decoding is required, the Reed-Solomon implementation is significantly slower. Hence, as the frequency of attacked pages increases, LT codes will eventually offer better performance than Reed-Solomon. Additionally, note that the Reed-Solomon and baseline "per skipped" encoding and decoding are approximately equal. This is to be expected, as both approaches are doing the same work (*i.e.,* computing the hash integrity check and nothing else).

Table 8.1
Summary of the performance impact for incorporating encoding into mode switches and performing on-demand frame decoding. All measured times are reported in microseconds.

Metric	Baseline	LT	RS
Time per mode switch	90.988	270.333	108.678
without encoding	3.161	3.232	3.174
Time per encoded frame	24.644	40.913	30.051
per full	50.627	n/a	483.745
per skipped	24.623	n/a	24.064
Time per decoded frame	25.785	35.640	26.044
per full	35.543	n/a	228.646
per skipped	25.845	n/a	25.401
Overhead relative to			
user mode time	4.1 %	11.8 %	4.6 %
total time	3.5 %	10.4 %	3.8 %

The other factor that influences the performance impact is the number of pages referenced during a single quantum. As this ratio increases, the amount of time spent decoding and encoding pages increases. However, the principle of locality tells us that, in a given time frame, most memory references will be from approximately the same region. That is, the number of pages referenced per quantum is typically small. In the case of *nbench*, we typically observed around 4 page decodings per mode switch. This ratio appears to be typical, based on other tests we ran. The maximum ratio of decodings per mode switch that we encountered was around 6 pages, which we observed in a custom application that generated random memory references. Thus, the numbers reported for *nbench* seem to be representative of typical applications. Hence, unless the system requires tight real-time guarantees in response to interrupts (which could be affected by the delay in mode switch processing), the performance impact of our approach is quite reasonable.

8.5.2 Recovery Success & Memory Overhead

Recall that LT codes provide only a probabilistic guarantee of data recovery. Figure 8.2 summarizes the success rates for $K = 16$ and various sizes of N. As our system is designed so that only 16 of the encoded blocks are to be stored in physical memory, we limit our evaluation to cases where some of these 16 blocks are corrupted. The results shown are the result of executing the LT encoding on a random block of 4096 bytes and randomly selecting blocks to discard. For each number of discarded blocks, we executed the encoding and decoding 10,000 times.

Table 8.2
Rate of successful data recovery for LT codes for $K = 16$ and various sizes of N

Number of	Rate of successful data recovery		
discarded blocks	$N = 18$	$N = 19$	$N = 20$
0	100.00 %	100.00 %	100.00 %
1	66.83 %	80.83 %	88.42 %
2	23.24 %	44.27 %	63.85 %
3	0.00 %	13.28 %	32.85 %
4	0.00 %	0.00 %	9.14 %
5	0.00 %	0.00 %	0.00 %

For $N = 18$, it is encouraging that we have more than a 50% chance of successful recovery for up to two corrupted blocks. As N increases, so do the chances of recovery. For $N = 20$, three blocks can be corrupted while still providing more than a 30% chance of recovery. The trade-off, though, is that $N = 18$ requires less than 20% memory overhead (as shown in Figure 8.3), while $N = 20$ would require approximately 32% memory overhead (to accommodate for the two additional blocks).

Our choice of $K = 16$ required consideration. Figure 8.3 summarizes the extra memory required to store the hashes and extra encoded blocks for various sizes of K. (For completeness, we note that the storage of the encoding graph (120 bytes) is trivial, since it is shared for all frames.) As shown in the figure, using fewer than $K = 16$ blocks would increase the memory overhead to more than 20%. On the other hand, using more than 16 blocks would increase the performance overhead, as the VMM would be required to spend more time computing the hashes of each block. Thus, $K = 16$ offers a balance that minimizes both the memory and performance overhead.

Table 8.3
LT storage overhead per page (in bytes) for various sizes of K, assuming $N = K + 2$.

Value of K	Hash overhead	Extra blocks	Total overhead	Percent of memory
8	180	1024	1204	29.39 %
16	340	512	852	20.80 %
32	660	256	916	22.36 %

To evaluate the probability of successful recovery with Reed-Solomon codes, we performed a probabilistic analysis, where the attacker randomly selects and corrupts a number of bytes. To simplify the analysis, we used a generating function based on the following polynomial:

$$P(z) = \left(\sum_{j=0}^{(n-k)/2} z^j \right)^{\left\lceil \frac{4096}{k} \right\rceil}$$

When this polynomial is expanded, the ratio of the coefficient of z^i to the sum of all coefficients provides an approximation of the probability that, given a random choice of i bytes, the attacker has successfully picked more than $(n-k)/2$ from one of the arrays. That is, this ratio is the probability that the attacker has successfully corrupted the page beyond our recovery technique. Using this technique, Figure 8.4 shows the cumulative probability that our approach can successfully recover given various parameters for k and the number of bytes corrupted. For instance, using Reed-Solomon(255,223), if the attacker corrupts less than 150 bytes in a single frame, our system can repair the damage with a probability of 52.78%.

Table 8.4
Rate of successful data recovery for Reed-Solomon(255,k) codes. Values marked with a * are not exactly 100% or 0%, but the difference is negligible (less than 1 in 1,000,000). The dashed lines show cut-offs that correspond to block sizes in the LT scheme (*e.g.,* 250 bytes in the Reed-Solomon scheme roughly correspond to one corrupted LT block).

Maximum number of corrupted bytes	Rate of successful data recovery		
	$k = 223$	$k = 207$	$k = 191$
50	100.00 %*	100.00 %*	100.00 %*
100	99.27 %	100.00 %*	100.00 %*
150	52.78 %	99.75 %	100.00 %*
200	1.12 %	88.88 %	99.97 %
250	0.00 %*	37.33 %	98.88 %
300	0.00 %*	3.01 %	87.47 %
350	0.00 %	0.02 %	51.33 %
400	0.00 %	0.00 %*	13.98 %
450	0.00 %	0.00 %*	1.34 %
500	0.00 %	0.00 %	0.03 %
550	0.00 %	0.00 %	0.00 %*
700	0.00 %	0.00 %	0.00 %*

Figure 8.5 summarizes the storage overhead per page for the Reed-Solomon version. Note that $n = 255$ is the only feasible first parameter, as $n = 2^m - 1$, where m is the number of bits in each symbol. Accommodating $m = 8$ would require too many bit manipulations to provide a feasible approach. For the second parameter, we consider k values of 223, 207, and 191, which allow for the correction of 16, 24, and 32 byte corruptions (per array), respectively. Obviously, as k decreases, we can provide better probability for successful recovery. However, only $k = 223$ allows us to keep the memory overhead below 20%.

8.6 Conclusion

The current state of modern computer systems requires the execution of trusted applications in untrusted environments. In this work, we have proposed a novel

Table 8.5
Reed-Solomon overhead (in bytes) per page.

(n, k)	Hash	Syn-drome	Flag	Total overhead	Percent of memory
(255,223)	20	608	1	629	15.36 %
(255,207)	20	960	1	981	23.95 %
(255,191)	20	1408	1	1429	34.89 %

approach for protecting applications from such attacks by incorporating efficient encoding of memory contents during the context switch procedure. In our approach, when the context switches from guest application A to B, the VMM encodes A's memory image, then decodes B's image after performing an integrity check. Unlike previous schemes, ours is unique in the sense that only ours offers the possibility that *the corrupted application can be repaired and allowed to continue execution.*

We presented both the design and a prototype implementation. Our empirical results show that both the memory and performance overhead imposed by our design is reasonable. One possible direction for future work would be to offer a more fine-grained approach to memory protection. For instance, it may be desirable to protect only critical portions, therefore reducing execution and protected storage overhead. Another direction would be to define the protocols and program analysis techniques for remote application recovery, as discussed in Section 8.4.8. Next, other types of encoding schemes could be explored. For instance, Raptor codes offer faster performance, but greater implementation complexity, and would be a good choice for additional evaluation. Finally, other VMM-based protection mechanisms offer additional protections that our scheme does not address. For example, Overshadow also provides confidentiality for the application. We view our scheme as complementary to these other solutions, and believe that combining them would create a very strong protection mechanism. Quantifying the costs involved in such a scheme would be an important step in evaluating such a system.

9 MINIMIZATION AND RANDOMIZATION AS DEFENSE AGAINST LIBRARY ATTACKS

In this chapter, we continue the discussion of execution integrity. While the preceding chapter began with a very powerful adversarial assumption, we consider a more modest and commonplace threat in this chapter. Specifically, we consider attacks on shared library code, such as return-into-*libc* and return-oriented programming. While code randomization has been used as a defense against these attacks, the way it has been applied is insufficient. In this discussion, we revisit the topic by proposing a fine-grained approach to randomization that overcomes the shortcomings of existing defenses.

9.1 The Lack of Sufficient Defense Against Library-based Attacks

The history of malware defense shows a clear pattern as an arms race between attackers and defenders. Attackers propose new techniques and defenders respond by finding methods to stymie exploits. While these defensive tools are being deployed, creative attackers find ways to bypass the schemes. Consider the history of buffer overflows [187, 188] and string format vulnerabilities [189]. The evolution of these attacks started with basic stack smashing then extended to other forms of corruption, such as heap-based code injection. At every step of the way, defenders found ways to violate an invariant of the attack behavior [91, 92, 190, 191]. For instance, canary words, instruction set randomization, and base address randomization either stopped attacks, or made crafting exploits more difficult. Attackers then responded with techniques to bypass the defenses [93, 192].

While earlier exploits involved the injection of malicious code, the recent trend has been to manipulate code that already exists, primarily in shared libraries such as *libc*. In a basic return-into-*libc* attack [193], for instance, a buffer overflow corrupts the return address to jump to a *libc* function, such as `system`. This type of attack then evolved into return-oriented programming (ROP) [194]. In ROP, the attacker identifies small sequences of binary instructions that end in a `ret` instruction. By placing a sequence of carefully crafted return addresses on the stack, the attacker can use these *gadgets* to perform arbitrary computation. These attacks continued to evolve, with newer techniques using gadgets that end in `jmp` or `call` instructions [195].

Early solutions to the problem of library-based exploits focused on the introduction of randomness into the memory image of a process. Specifically, by randomizing the start address of the code segment or mapped library, a single exploit packet would not be effective on all running instances of an application. That is, two different running instances would have a different base address, so the addresses that an attacker needed to jump to in one instance would not be the same as the addresses in the other instance. Although randomization initially seemed promising, these solutions suffered by the small amount of randomization possible [94]. For instance, there were only 2^{16} possible start addresses for 32-bit machines. Consequently, successful brute-force attacks were feasible. The solution proposed by many researchers was simply upgrade to 64-bit architectures.

We find the notion that upgrading is the only solution to be unsatisfactory. First, there are many cases where it is not feasible. Specifically, embedded systems often have strict requirements for simplicity and power consumption. Using 64-bit architectures is simply not an option. Second, expecting the world's entire 32-bit user community to immediately upgrade is extremely naïve. The problem is not just the cost of hardware. Rather, upgrading the hardware for mission-critical systems, for example, requires a significant amount of planning, evaluation, and documentation. It is impractical to demand enterprises to undertake this process in response to an attack vector that, at the time of this writing, is dwarfed by other threats. In short, we see no reason to think that 32-bit systems will be retired any time soon, despite the existence of attacks in the wild. Third, and most problematic, upgrading does not necessariliy solve the problem. Recent work has demonstrated that an attacker can use information leakage to discover the randomization parameters, thus negating the promised benefits [95].

Rather than abandoning the idea of randomization on 32-bit architectures, we propose to re-examine the granularity at which it is performed. We start with the observation that applications rarely, if ever, use the full code base of shared libraries. Rather, for any two applications, there is a strong likelihood that the subset of *libc* code needed is different. Consequently, mapping only the library functions needed for a process will produce a different memory image for each application. Second, the amount of possible randomization generated can be further increased by permuting the code blocks within the library. For instance, if an application uses 500 of the code symbols in *libc*, there are $500! \approx 2^{3767}$ possible permutations of the code blocks, which obviously exceeds the 2^{16} base addresses by leaps and bounds.

This minimize-and-randomize approach has many benefits. First, as stated above, the number of possible randomized results clearly makes brute-force approaches infeasible. Next, and more subtly, the code required for an attack *may not exist in memory*. If none of the code that makes up the `system` library function, for instance, is required for an application, it is simply not mapped into memory; consequently, there is no way for a return-into-*libc* exploit to jump to it. Finally, our scheme offers an alternative to approaches that dynamically monitor critical data like return addresses. Although these schemes are effective, they distribute the performance cost throughout the execution life-time of the process. In our solution, the entire performance cost is paid once during process setup, and is quite reasonable; after the execution begins, the code runs as originally designed.

With any solution, there are always costs that must also be considered. In our proposed scheme, there is a performance impact when the process begins. In Section 9.4 we describe techniques for minimizing this impact. Another cost of our solution is reduced sharing among processes. As every process will have a different image of *libc* in memory, then the pages for this code can not be shared; this increases the total memory consumption for applications. We note, though, that this impact could be reduced for multiple processes of the same application. That is, one could randomize the library only if no instances of the application are running; if another process for the application already exists, the loader could reuse the randomized pages. Clearly, reusing pages in this way decreases the randomization, but may be an acceptable trade-off. Thus, fine-grained randomization is a mixed blessing, as we are trading off increased memory consumption for safety through diversity.

Finally, there is still a potential attack vector with our scheme. Specifically, once the library is randomized, the addresses are stored statically in the global offset table (GOT). Thus, there is a clear source of information leakage. However, we note that, with high probability, the attacker would have to read significant portions of the GOT at run-time, *prior to* crafting the attack payload. While such a feat is not impossible, we intuit that it is significantly more difficult than existing strategies.

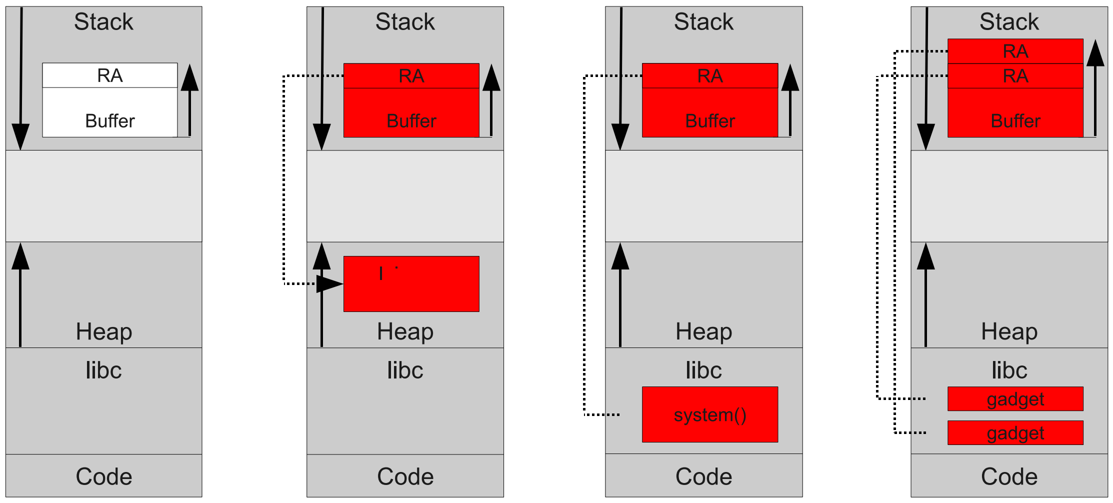
9.2 Background

The focus of our work is on attacks that use existing code stored in shared libraries. In this section, we start with a brief summary of these attack techniques. Figure 9.1 shows the evolution of buffer overflow attacks. After describing the evolution of attacks, we then summarize critical enabling factors that guide our defensive technique design.

9.2.1 Library-based Attacks

Return-into-*libc* [193] attacks are a special class of stack-based exploits.[1] While buffer overflows traditionally used the corrupted return address to jump to an address on the heap (where injected code was placed), return-into-*libc* attacks jumped to existing code in the *libc* shared library. Frequently, the address of choice was the

[1]Note that return-into-*libc* attacks are not limited to just stack-based buffer overflows. That is, an attacker can corrupt a data pointer on the heap and use it to modify the return address without an explicit overflow on the stack.



(a) Original layout (b) Injected code overflow (c) Return-into-*libc* attack (d) ROP attack

Figure 9.1. Evolution of buffer overflow attacks, from stack-smashing with injected code to gadget-oriented programming

location of the `system` function, which allowed the attacker to perform an arbitrary system call or spawn a shell. As return-into-*libc* exploits involve corrupting a return address to jump to a known location in *libc* for malicious purposes, they can be seen as a special case of return-oriented programming (ROP).

In ROP exploits, an attacker crafts a sequence of *gadgets* that are present in existing code to perform arbitrary computation. A gadget is a small sequence of binary code that ends in a `ret` instruction. By carefully crafting a sequence of addresses on the software stack, an attacker can manipulate the `ret` instruction semantics to jump to arbitrary addresses that correspond to the beginning of gadgets. Doing so allows the attacker to perform arbitrary computation. These techniques work in both word-aligned architectures like RISC [196] and unaligned CISC architectures [194]. ROP techniques can be used to create rootkits [197], can inject code into Harvard architectures [198], and have been used to perform privilege escalation in Android [199]. Initiating a ROP attack is made even easier by the availability of architecture-independent algorithms to automate gadget creation [200].

While researchers were exploring defenses against return-oriented attacks, similar techniques can manipulate other instructions, such as `jmp` and their variants [195,201, 202]. While the semantics of the gadgets differ from ROP techniques, jump-oriented techniques are built on the same premise: By stringing together a sequence of small gadgets, the attacker can perform arbitrary computation without code injection. To simplify the discussion, we generalize these techniques under the term gadget-oriented programming (GOP).[2] Note, though, that return-into-*libc* attacks do not use gadgets. Consequently, we use the collective term *library-based attacks* to refer to all of these techniques.

While GOP techniques are primarily studied for their applications in malware and execution corruption, other interesting applications have also been explored. Specifically, GOP can be used to create platform-independent programs [203] (PIP). A PIP is a binary image that will execute without error on two or more architectures without modification. One useful scenario for PIPs is program steganography. That is, a PIP could be created and explicitly identified as a Windows x86 `.exe` file, while the intended (secret) computation is achieved by running the program on an ARM platform.

9.2.2 Attack Behavior Summary

Based on our survey of library-based attacks and defenses, we have identified a number of distinct characteristics and requirements for a successful exploit. We argue that a defensive technique that undermines these invariants will present a robust protection mechanism against these threats. We summarize the fundamental assumptions and enabling factors as follows.

- If an application requires access to a single function in a shared library, then the full code base of the library is mapped into the process memory image.

[2]Such a general term has the advantage that, if someone proposes a similar technique based on, say, `call` instructions, a new name is not necessary.

- The relative offsets of code within the mapped library are constant. That is, if an attacker knows the base address of the mapped library, then the location of all gadgets and symbols is deterministic.
- The same sequence of gadgets can be reused for exploits in different applications. That is, once an attacker has identified an exploitable vulnerability in multiple applications, writing the same payload of return addresses to the stack will produce the same attack behavior.

The first of these factors is a problem of over-provisioning of access. That is, the coarse granularity of shared library loading is inadequate for enforcing the *principle of least privilege* within the library. Consequently, an attacker that can exploit an application vulnerability has more access to the library than is necessary. By minimizing the mapped image to only the required functionality, the loader enforces least privilege for code more accurately.

The second factor contributes to the weakness of ASLR on 64-bit architectures. Recall that information leakage allows an attacker to learn the randomized base address at run-time [95]. As a result, once this address is known, the attacker can construct a library-based attack using the known relative offsets. Shuffling the code within the library at run-time raises the bar significantly. Instead of learning a single address, the attacker would have to locate the GOT (which may be in a randomized location) and read its contents at run-time *prior to* constructing the stack-based payload. Thus, fine-grained randomization is a substantial advance in the protection mechanism.

The third factor, which follows from the first two, illustrates an additional protective feature of Marlin. With very high probability, any two given applications will require a different subset of symbols in the shared library. As such, the gadgets available in the process image for one application may not be present in the memory image of the other application. Consequently, the attacker would have to ensure that the vulnerable application actually used the symbols containing the desired gadgets.

In addition, the run-time shuffling of the symbols would prevent multiple instances of the same program from having the same library layout. Specifically, there are $n!$ possible permutations for n symbols, making a brute-force attack on the randomizations infeasible. Thus, a defensive technique that prohibits this factor would require an attacker to construct a new exploit *for every instance of every application.*

9.3 *libc* Analysis

In designing our solution, we performed an empirical analysis of the usage of the *libc* library by 51 popular applications, which are listed in the Appendix. To better understand typical *libc* usage in a variety of settings, we selected applications from a broad range of categories (*e.g.,* network applications, language interpreters, virtual machine monitors, media applications). Our analysis is based solely on the binary executable, so we considered both open- and closed-source software. All software packages were downloaded for Ubuntu 10.04 ("Lucid Lynx") for version 2.6.32 of the Linux kernel.

To analyze an application's *libc* usage, we performed an any-path evaluation, beginning with the set of dynamically mapped symbols in the program's ELF symbol table, retrieved with the `objdump` utility. That is, we maintained a working set of symbols to evaluate, where the set was initialized with the application's required symbols. We processed each symbol by traversing the corresponding binary instructions in a disassembled version of *libc*. If we encountered any variation of a `ret`, `jmp`, or

Table 9.1
Statistics concerning lines of assembly code, symbols, and branching instructions (`ret`, `jmp`, `call`) in *libc*

	LOC	Symbols	Returns	Jumps	Calls
Total	276,970	2171	3472	39,085	12,828
Min	131,653	360	1013	19,172	4762
Average	145,498	450	1188	20,985	5423
Std Dev	13,350	75	159	1782	626

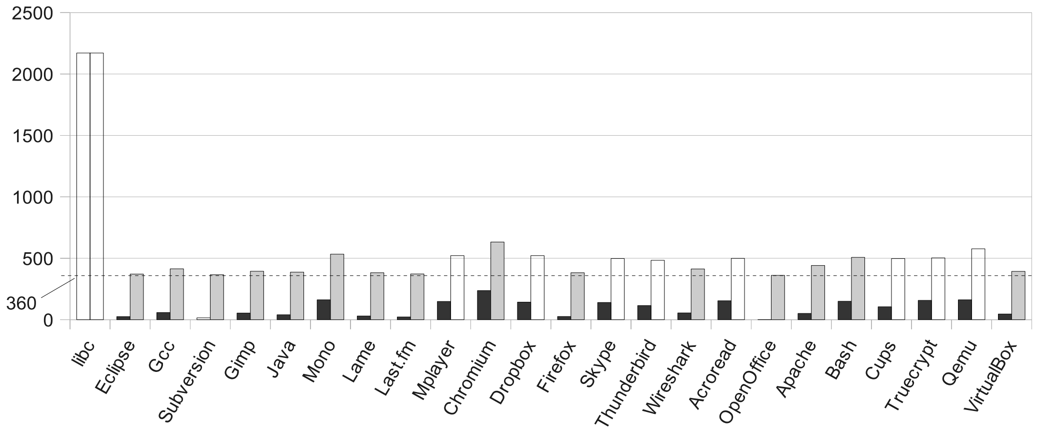


Figure 9.2. Number of *libc* symbols used by popular applications

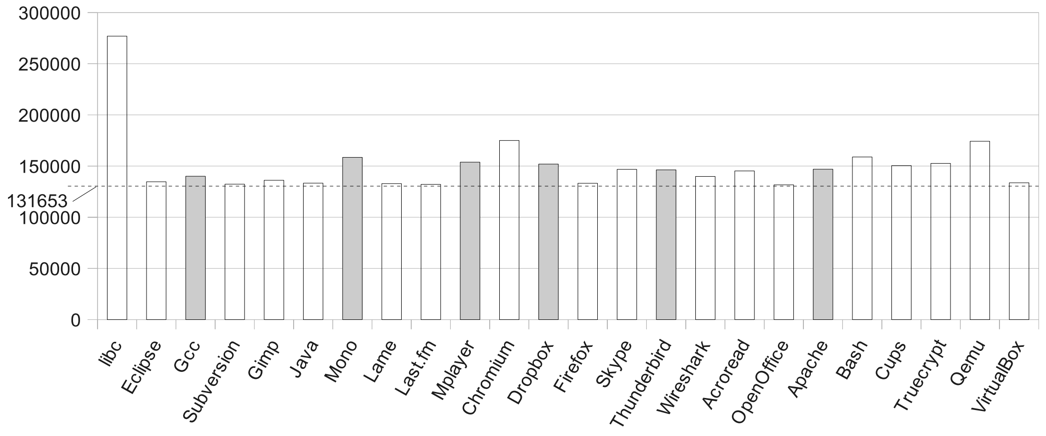


Figure 9.3. Number of lines of *libc* assembly code used by popular applications

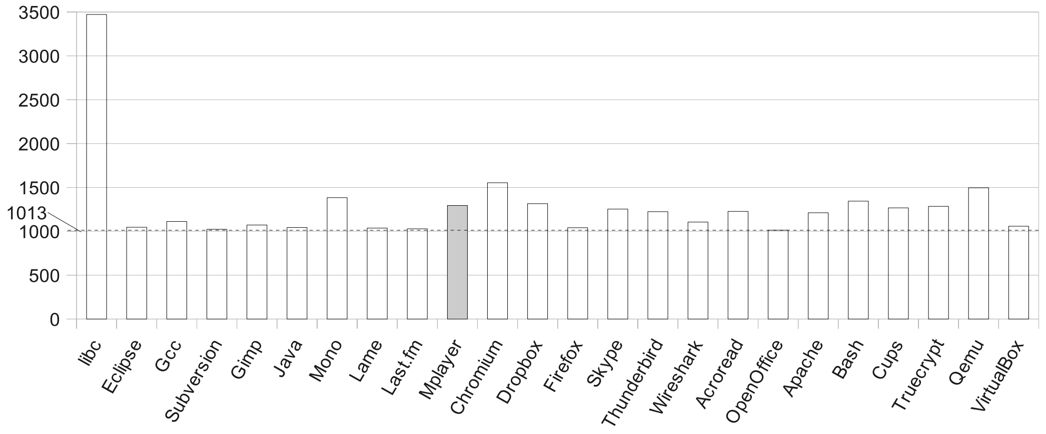


Figure 9.4. Number of returns in *libc* code used by popular applications

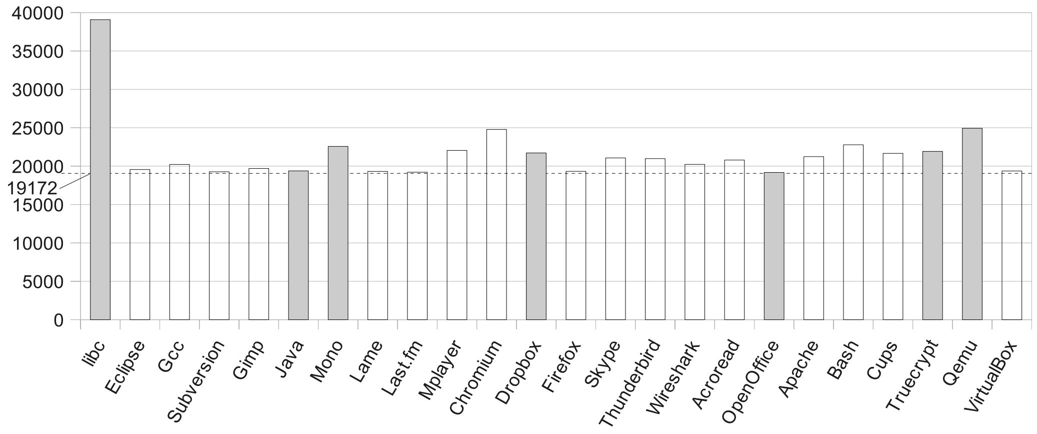


Figure 9.5. Number of jumps in *libc* code used by popular applications

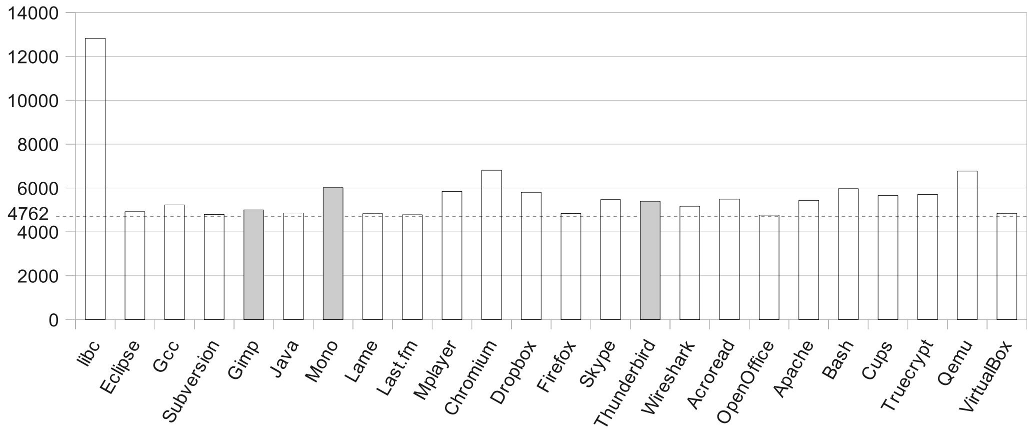


Figure 9.6. Number of calls in *libc* code used by popular applications

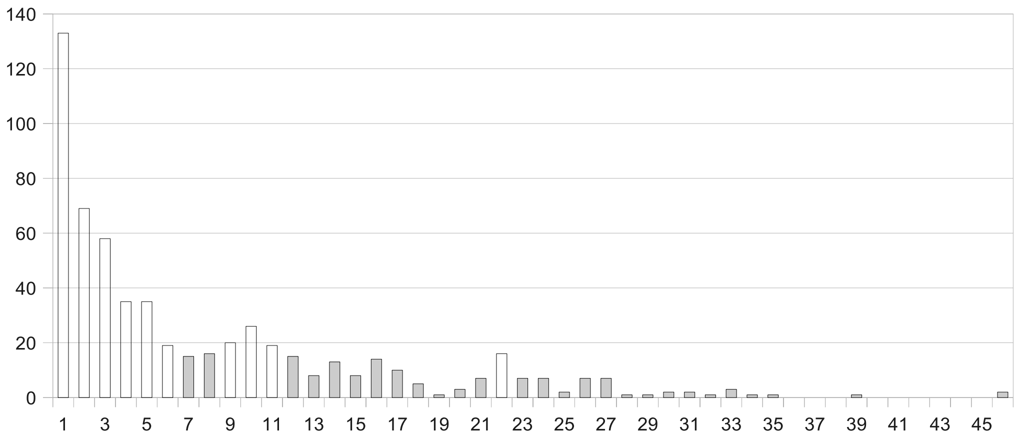


Figure 9.7. Histogram of non-minimum symbols used by popular applications

`call` instruction, we added the target symbol to the working set. We proceeded in this manner until we reached the least fixed point of symbol references, indicating that we had traversed all *libc* instructions that could possibly be reached from the symbols in the application's symbol table.

In our static analysis, we discovered that there is a significant minimum threshold of *libc* code that must be mapped for all applications. Specifically, all processes created from ELF executables jump to the `__libc_start_main` symbol, which indicates the start of the program. From this single symbol, every application reaches 360 of the 2171 function symbols in *libc*. Figure 9.1 shows the total and minimum *libc* code base, as well as some basic statistics about the applications we surveyed. In Figures 9.2-9.6, the minimum values are labeled and marked with dotted line for reference.

There are some interesting lessons to take from Figure 9.1. First, the minimum of 360 symbols ensures that, if the entire library is permuted at run-time, there is no shortage of randomization. That is, in contrast with the 2^{16} possible layouts that can be generated by randomizing the segment base address (*e.g.,* as in PaX), shuffling the library guarantees a minimum of $360! \approx 2^{2543}$ possible permutations. Second, the average application requires only 90 additional symbols and introduces only 175 `ret`, 1813 `jmp`, and 661 `call` instructions, when compared with the minimum. That is, if the attacker is restricted to constructing gadgets only from the additional code base, the number of gadgets in memory may be too small to achieve the desired behavior. Finally, note that minimizing the *libc* code according to the application's needed functionality has a drastic effect on the attack surface. On average, the lines of code (*i.e.,* assembly instructions) and branching instructions are significantly smaller than the full library size. In other words, mapping the full library gives the attacker access to a significant amount of unused code for gadget construction.

Figure 9.2 shows the number of *libc* symbols used by a subset of the applications we surveyed. For each application, the darker region on the left indicates the number of symbols explicitly identified by performing `objdump` on the executable; the ligher region on the right denotes the total number of symbols reached through our any-path analysis. Figure 9.3 shows the total number of *libc* lines of code that are required for each application. This figure shows that most applications only require about half the code that exists in *libc*. Furthermore, observe that most of this code is the minimum (*i.e.,* the code reachable from `__libc_start_main` entry point.

Figures 9.4, 9.5, and 9.6 show the reduced attack surface for gadget construction. Specifically, the number of `ret` instructions is cut down to a third, from 3472 in *libc* down to an average of 1188 (see Figure 9.1). Similarly, `jmp` and `call` instructions are cut to about half, from 39,085 to 20,985 and from 12,828 to 5423, respectively.

Finally, Figure 9.7 shows an interesting result, the frequency of non-minimal symbol usage among the 51 applications surveyed. That is, setting aside the 360 symbols derived from `__libc_start_main`, we calculated how many of the applications used each of the remaining symbols. For instance, there were 133 symbols that were used by only one application, while there was only one symbol used by 39 applications. Furthermore, 426 (72.2%) of the symbols in *libc* were used by 10 or fewer applications, while 522 (88.5%) of the symbols were used by 20 or fewer applications. In other words, there is significant variation in the non-minimal symbols that are used by applications. Consequently, the probability of any two applications using exactly the same set of symbols is very small; we observed only a single pair of applications (`gs` and OpenOffice) that used the exact same set of symbols, which happened to be the minimal set.

9.4 Marlin

In this section, we describe the design of Marlin, our defense against library-based attacks. Marlin is, primarily, a customized loader augmented with application control-flow analysis. We start this section by describing the basic design of Marlin. We then describe optimizations that help to reduce the performance impact of Marlin.

9.4.1 Control-flow Analysis

As in Section 9.3, any-path control-flow binary analysis lies at the heart of Marlin. To prevent unnecessary computation, Marlin generates and stores the resulting control-flow graph. Then, when an application is run, Marlin's loader reads the ELF header information and creates a starting set of the dynamically mapped symbols, as well as the 360 minimal symbols (which are reached from `__libc_start_main` and used by all processes). Marlin then examines the control-flow graph to determine the set of reachable symbols. A straightforward performance optimization would be to perform this analysis once for each application, storing the result in a database maintained by the loader. The database would only need updated when the library or application code change.

9.4.2 Code Randomization

The control-flow analysis produces, at run-time, the complete set of shared library symbols required by the new process. Marlin then generates a random permutation of this set of symbols. The resulting permutation determines the order in which the `mmap` system calls are issued, which changes the order of the mapped symbols in memory. The drastically increased number of `mmap` calls is the primary performance hit incurred by Marlin. However, as we will show in Section 9.5, these calls are very efficient, and the cumulative performance impact is reasonable.

One may object that we are not providing a full accounting of the performance impact of Marlin's library randomization. Specifically, by shuffling the library, Marlin may have an impact on the principle of locality by increasing the distance between code that would normally be in close proximity. As such, it is conceivable that Marlin would induce a larger number of page faults, thereby incurring a hidden performance penalty. However, functions tend to be self-contained units and it is not apparent how large the impact would be, as hardware and system configuration also influence the page fault rate. At this point, we have not been able to quantify this effect, and leave this issue open for future consideration.

9.4.3 Minimum Code Optimization

Recall that there are 360 *libc* symbols (derived from the `__libc_start_main` symbol) that are used by all programs. One optimization of Marlin is to eliminate these symbols from the main pool of permuted *libc* symbols. That is, since applications use an average of 450 *libc* symbols, we generate a set of the 90 addition symbols. Observe that this produces $90! \approx 2^{459}$ possible permutations at run-time, which far exceeds the 2^{16} possible randomizations introduced by ASLR.

The handling of the 360 minimum symbols requires careful consideration. Recall from Figure 9.1 that this set contains 1013 `ret`, 19,172 `jmp`, and 4762 `call` instructions. While we have not examined these instructions in detail, our intuition is that this code base is sufficiently large to generate enough gadgets for an attack. As such, if we were to leave these symbols as a single block that remains identical, the security guarantees provided by Marlin would be weakened. On the other hand, it would be very desirable to make this set re-usable across multiple processes to reduce the memory cost and performance impact of Marlin.

As a middle ground, our approach is to generate a single permutation of the minimum symbol set at run-time, mapping this block of symbols to a contiguous page-aligned portion of memory.[3] The pages that correspond to this block can then be shared by running processes as needed. Observe that this approach produces $360! \approx 2^{2543}$ permutations, one of which is selected randomly at system boot. Consequently, the probability that any two running systems will share the same permutation of the 360 minimum symbols is negligible. As such, the ability to apply the same exploit payload on multiple systems would be virtually eliminated.

[3]To prevent this sharing from become a vulnerability, Marlin can regenerate the minimum symbol set permutation at regular intervals during execution. That is, running processes would continue to use the previous permutation, but new processes would use a different memory image.

9.4.4 Application Pre-processing

The impact of the non-minimal code randomization can be reduced even further by taking the permutation generation off-line. To do so, each application will have a dedicated file containing the next instance's permutation. When a new process is created, the loader sends a signal to a trusted daemon process that runs with low priority. This process then shuffles the application's library map accordingly.

9.5 Prototype Implementation

We have implemented a preliminary Marlin prototype as a proof-of-concept. Our implementation is built on version 11.02 of the Genode OS framework, running on top of the L4Ka::Pistachio microkernel. We opted for this software stack, as our desire for absolute minimization is more consistent with the microkernel philosophy. We modified the Genode shared library loader (`ldso`) and built a QEMU virtual machine instance.

```
[init -> test-lib] "FunctionA" in "binary" ==> 00019180
    in "testlib.lib.so"
[init -> test-lib] reloc_jmpslot:  *00323cd8=00019180
[init -> test-lib] FunctionA: 35
[init -> test-lib] &FunctionA: 0031b7e0
[init -> test-lib] &FunctionB: 00019190
[init -> test-lib] FunctionB: 800
```

Figure 9.8. Output from mapping both library functions

Figures 9.8 and 9.9 show the output from a stack-smashing attack with and without Marlin. To clearly illustrate the functionality and protection, we generated a sample shared library (`testlib.lib.so`) that consisted of only two functions (`FunctionA` and `FunctionB`). The first line of Figure 9.8 shows the entire library being mapped (without Marlin) at memory address `0x00019180`. The next line shows the use of a jump table for the dynamic mapping. That is, while the actual library code begins at

0x00019180, 0x00323cd8 is the address of the GOT entry that points to FunctionA. The third line shows the output of calling FunctionA (it just prints 35).

The next two lines show addresses for FunctionA and FunctionB, but may seem misleading. While the GOT stores the addresses of the dynamically linked functions, another table is required. Specifically, the procedure linkage table (PLT) stores a sequence of small dispatch routines for dynamically linked functions. In this case, 0x0031b7e0 stores the location of a wrapper routine that finds and jumps to the address of FunctionA in the GOT. On the other hand, 0x00019190 is the address of the code for FunctionB, which we compute directly without using the PLT. Finally, our exploit corrupts the stack to place the address of FunctionB where the return address is stored. As a result, when the ret instruction is executed, control jumps to 0x00019190 and FunctionB executes, producing the last line of output (FunctionB just prints 800).

```
[init -> test-lib] "FunctionA" in "binary" ==> 00019180
    in "testlib.lib.so"
[init -> test-lib] reloc_jmpslot:  *00323cd8=00019180
[init -> test-lib] FunctionA: 35
[init -> test-lib] &FunctionA: 0031b7e0
[init -> test-lib] &FunctionB: 00019190
attempted write at read-only memory (WRITE pf_addr=0001919c
    pf_ip=00019190 from 601 (raw 01804002))
```

Figure 9.9. Output from mapping both library functions

Figure 9.9 shows the same exploit, but with Marlin enabled for an application that only requires FunctionA. That is, FunctionB is *not* mapped into memory. In this figure, one can see that the last line of Figure 9.8 (showing the execution of FunctionB) is not produced. Instead, an error message is generated, as the address where FunctionB should be contains whatever random data was previously stored in that memory location. That is, *the attacker can no longer use the address of* FunctionB *for an attack.*

To evaluate the overhead of our prototype, recall that the primary performance impact of Marlin is an increase in the number of `mmap` calls made by `ldso`. Our test machine consisted of a 2.4 GHz Core 2 Duo CPU with 4 GB of DDR2 memory, running Mac OS X 10.6. Our Marlin prototype ran as a QEMU 0.13.0 virtual machine instance with 130 MB of memory. The original Genode loader took an average of 0.2639 seconds to run, with an average execution time of 0.0057 seconds per `mmap` call. Recall that Figure 9.1 showed the average application required 450 *libc* symbols. Based on this data, we can extrapolate that the average overhead induced by Marlin would be adding 2.565 seconds to each application's startup time, which is essentially an increased order of magnitude for the loader execution time.

While this performance impact may be cumbersome, we counter objections with three points. First, this overhead is a one-time cost for each process. Once the process is created, execution proceeds like normal, with no additional penalty at run-time. Second, these measurements were taken within a virtualization environment, implying that the performance overhead on native hardware would be significantly reduced. Finally, in Section 9.4.3, we described an optimization wherein the 360 required *libc* symbols are mapped once and shared by all processes. Consequently, a new process would only need to map 90 additional symbols, incurring a 0.513 second overhead to startup time. Based on the significantly increased randomization that Marlin creates, we argue that a one-time penalty of half a second is clearly a reasonable performance impact for a robust defense against library-based attacks.

9.6 Discussion

When we first discussed the idea of this work to other security researchers, the consensus seemed to be that it was unnecessary. Specifically, the problem with ASLR was simply the small number of possible 32-bit base address randomizations, and upgrading to 64-bit architectures was the solution. However, as shown in [95], the security of this solution is not guaranteed. Consequently, our goal for this work was to re-open the discussion of how to randomize application memory images to defend against library-based attacks.

In designing our solution, we analyzed how much of *libc* is actually required for an application. We found that there is a substantial portion (360 symbols, representing 131,653 lines of assembly code) that is required for all applications. However, this corresponds only to 16.6% of the symbols and 47.5% of the code, and mapping the entire library was equivalent to over-provisioning access for potential exploit. Other than this minimum code base, the actual usage of *libc* code varies greatly by real applications. Consequently, we wanted to use this diversity as a new foundation for run-time library randomization. We subsequently showed that fine-grained randomization of libraries produces an extremely large number of possible permutations, effectively making brute-force de-randomization impossible.

As we mentioned previously, once the randomized mapping has been generated, there is still the possibility that an attacker could de-randomize the addresses by reading the GOT. That is, we did not want to fall victim to the same attack described in [95]. We argue that we have done so. Specifically, in that work, the attacker only needed to determine a single piece of data: the randomized base adress. This information, coupled with background knowledge of the structure of *libc*, allows an attacker to determine the location of the `system` function or construct a chain of gadgets.

With Marlin, however, the attacker must locate and *traverse* the GOT. Next, the exploit payload must be capable of combining the information gathered from this traversal with the location of gadgets or the coveted `system` function. However, *this implies that the payload is already more powerful than existing exploits.* Specifically, in a current attack, the payload is simply a piece of data used for a buffer overflow; on the other hand, the payload in an attack on Marlin must be able to read the GOT data and branch accordingly (*i.e.,* if this entry is `system`, do `X`, else continue searching). Alternatively, if the application is a server, the attacker could use one exploit to leak the data, perform the analysis off-line, and craft the resulting buffer overflow payload accordingly. While we do not see that this is a feasible attack strategy, we err on the side of caution and do not claim that this threat vector is impossible. Instead, we simply argue that *Marlin's run-time shuffling of dynamically mapped library symbols significantly increases the amount of randomization and, therefore, the security guarantees over current approaches.*

Finally, astute readers may note that we have neglected one important issue when it comes to GOP attacks: The number of gadgets available for an attack is determined by the size of the exposed code base. That is, the use of *libc* and other shared libraries is simply a matter of convenience, not a requirement; an attacker could simply use the application's binary image itself as the source of gadget code. As a means of comparison, our compiled version of *libc* is 1.3MB while our compiled version of Chromium is 46.4MB. Our intuition is that the Chromium code base is sufficiently large for a would-be attacker. Thus, our current solution does not entirely solve the problem of GOP attacks. As such, we argue that future work in this area should consider how to randomize the application binary image, as well.

9.7 Conclusions

In this work, we have revisited the idea of randomization as defense against library-based attacks. Specifically, as a defense against return-into-*libc* and return-oriented or jump-oriented programming (*i.e.,* gadget-oriented programming) attacks, we propose a fine-grained approach to shared libary randomization. Furthermore, we fully incorporate the principle of least privilege by mapping only the portions of the library that are required for the specific application. We have described the static analysis and dynamic mapping features underlying Marlin. In addition, we have demonstrated a proof-of-concept prototype, including an illustration that our approach makes the success of these attacks unlikely. We have shown that the cost of our defensive technique occurs as a single performance hit when the application boots; once the application is running, no additional overhead is incurred. Based on the results of our analysis and implementation, we argue that fine-grained library mapping is both feasible and practical as a defense against these pernicious library-based attack techniques.

9.8 Applications Surveyed

Application	**Version**	**Application**	**Version**
Acroread	9.3.2	Amazon MP3 Downloader	1.0.9
Apache	2.2.14	Bash	4.1.5
Bluetooth	4.60	Brasero	2.30.2
Chromium	10.0.648.205	Cups	1.4.3
Dhclient	3.1.3	Dropbox	0.6.7
Eclipse	3.5.2	Empathy	2.30.3
Evolution	2.28.3	Firefox	3.6.16
Gcc	4.4.3	Gimp	2.6.8
Git	1.7.0.4	Gnome-terminal	2.30.2
Grip	3.3.1	Gs	8.71
Gtkpod	0.99.14	Gzip	1.3.12
Java	1.6.0_20	Lame	3.98.2
Last.fm	1.5.4	*libc*	2.11.01
Make	3.81	Mencoder	4.4.3
Mono	2.4.4	Mplayer	1.0
OpenOffice	3.2.0	OpenSSH	5.3p1
PdfTeX	1.40.10-2.2	Perl	5.10.1
Python	2.6.5	Qemu	0.12.3
Sha1sum	7.4	Skype	2.1.0.81
Smbclient	3.4.7	Subversion	1.6.6
Sudo	1.7.2p1	Tar	1.22
Thunderbird	3.1.8	Totem	2.30.2
Truecrypt	6.3a	Vim	7.2
VirtualBox	4.0.0	Vlc	1.0.6
Transmission	1.93	Wine	1.2.2
Wireshark	1.2.7	Xine	0.99.6

10 SUMMARY

Establishing a trusted basis for CDAC requires addressing a number of challenges related to the enforcement mechanisms used. First, one must define the contextual factors that are relevant; additionally, one must specify what level of authentication is necessary to say that the contextual claim is trustworthy. Next, one needs to examine the relevant technologies and policy goals of the CDAC deployment. Finally, the architectures, protocols, and execution environments must be proposed and analyzed.

In this work, we have explored the design of multiple CDAC enforcement mechanisms for a variety of settings. We have demonstrated how to apply NFC technology to the problem of spatially aware RBAC, and shown how to consider other user's locations and to respect individual privacy concerns. Our work on PUFs has shown the feasibility of identifying the unique hardware instance prior to considering the access request, and established a hardware guarantee to restrict the number of times a cryptographic key can be used. Finally, have also designed novel approaches to ensuring the integrity of a trusted application by detecting memory corruption, as well as potentially repairing the application without interrupting the execution.

In examining these scenarios and evaluating our implementation work, it is clear that existing technologies make the deployment of CDAC systems possible. The proper combination of hardware, cryptography, and software can provide a highly assured root of trust that ensures the CDAC policies are enforced correctly. Furthermore, these guarantees hold even in the presence of malicious adversaries, including rogue trusted insiders.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Ravi Sandhu, Kumar Ranganathan, and Xinwen Zhang. Secure information sharing enabled by trusted computing and PEI models. In *Proceedings of the 1st ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 2–12, New York, NY, USA, 2006. ACM.

[2] Robert K. Merton. *On the Shoulders of Giants: A Shandean Postscript*. University of Chicago Press, 1993.

[3] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. Fuzzy MLS: An experiment on quantified risk-adaptive access control. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 222–230, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Nguyen Ngoc Diep, Le Xuan Hung, Yonil Zhung, Sungyoung Lee, Young-Koo Lee, and Heejo Lee. Enforcing access control using risk assessment. In *Proceedings of the 4th European Conference on Universal Multiservice Networks*, pages 419–424, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Nathan Dimmock, András Belokosztolszki, David Eyers, Jean Bacon, and Ken Moody. Using trust and risk in role-based access control policies. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 156–162, New York, NY, USA, 2004. ACM.

[6] Benjamin Aziz, Simon N. Foley, John Herbert, and Garret Swart. Reconfiguring role based access control policies using risk semantics. *Journal of High Speed Networks, Special Issue on Security Policy Management*, 15(3):261–273, 2006.

[7] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 1–10, New York, NY, USA, 2004. ACM.

[8] Jaehong Park and Ravi Sandhu. The UCON$_{ABC}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, February 2004.

[9] Xinwen Zhang, Masayuki Nakae, Michael J. Covington, and Ravi Sandhu. A usage-based authorization framework for collaborative computing systems. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 180–189, New York, NY, USA, 2006. ACM.

[10] Organization for the Advancement of Structured Information Standards (OASIS). eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml/`.

[11] Ravi Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[12] David Ferraiolo and Richard Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, October 1992.

[13] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *Proceedings of the 4th European Symposium on Research in Computer Security*, ESORICS '96, pages 65–79, London, UK, 1996. Springer-Verlag.

[14] Ravi Sandhu. Role activation hierarchies. In *Proceedings of the 3rd ACM Workshop on Role-based Access Control*, RBAC '98, pages 33–40, New York, NY, USA, 1998. ACM.

[15] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Frode Hansen and Vladimir Oleschuk. SRBAC: A spatial role-based access control model for mobile systems. In *Proceedings of the 8th Nordic workshop on Secure IT Systems*, NORDSEC '03, pages 129–141, Gjøvik, Norway, October 2003.

[17] Subhendu Aich, Shamik Sural, and Arun K. Majumdar. STARBAC: Spatiotemporal role based access control. In *OTM Conferences*, volume 4804 of *Lecture Notes in Computer Science*, pages 1567–1582. Springer, 2007.

[18] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information and System Security*, 10(1), February 2007.

[19] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A location-aware role-based access control model. In *Proceedings of the 2nd International Conference on Information Systems Security*, ICISS '06, pages 147–161, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] Suroop Chandran and James Joshi. LoT RBAC: A location and time-based RBAC model. In *Proceedings of the 6th Internation Conference on Web Information Systems Engineering*, WISE '05, pages 361–375. Springer-Verlag, 2005.

[21] Vijayalakshmi Atluri and Soon Ae Chun. A geotemporal role-based authorisation system. *International Journal of Information and Computer Security*, 1(1/2):143–168, January 2007.

[22] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dev, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, SACMAT '01, pages 10–20, New York, NY, USA, 2001. ACM.

[23] Maria Luisa Damiani and Elisa Bertino. Access control and privacy in location-aware services formobile organizations. In *Proceedings of the 7th International Conference on Mobile Data Management*, MDM '06, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Claudio Ardagna, Marco Cremonini, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Access control in location-based services. In Claudio Bettini, Sushil Jajodia, Pierangela Samarati, and X. Wang, editors, *Privacy in Location-Based Applications*, volume 5599 of *Lecture Notes in Computer Science*, pages 106–126. Springer Berlin, Berlin, Heidelberg, 2009.

[25] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The Cricket location-support system. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 32–43, New York, NY, USA, 2000. ACM.

[26] Michael S. Kirkpatrick and Elisa Bertino. Enforcing spatial constraints for mobile rbac systems. In *Proceeding of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, pages 99–108, New York, NY, USA, 2010. ACM.

[27] Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, and Kami Vaniea. Lessons learned from the deployment of a smartphone-based access-control system. In *Proceedings of the 3rd ACM Symposium on Usable Privacy and Security*, SOUPS '07, pages 64–75, New York, NY, USA, 2007. ACM.

[28] Naveen Sastry, Umesh Shankar, and David Wagner. Secure verification of location claims. In *Proceedings of the 2nd ACM Workshop on Wireless Security*, WiSe '03, pages 1–10, New York, NY, USA, 2003. ACM.

[29] Paramvir Bahl and Venkata N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *Proceedings of the 19th Annual Conference on Computer Communications*, INFOCOM '00, pages 775–784, Washington, DC, USA, 2000. IEEE Computer Society.

[30] Claudio A. Ardagna, Marco Cremonini, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Supporting location-based conditions in access control policies. In *Proceedings of the 1st ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 212–222, New York, NY, USA, 2006. ACM.

[31] Gerald Madlmayr, Josef Langer, Christian Kantner, and Josef Scharinger. NFC devices: Security and privacy. In *Proceedings of the 2008 3rd International Conference on Availability, Reliability and Security*, ARES '08, pages 642–647, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Collin Mulliner. Attacking NFC mobile phones. In *EUSecWest 2008*, May 2008.

[33] Elisa Bertino, Claudio Bettini, and Pierangela Samarati. A temporal authorization model. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, CCS '94, pages 126–135, New York, NY, USA, 1994. ACM.

[34] Devdatta Kulkarni and Anand Tripathi. Context-aware role-based access control in pervasive computing systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 113–122, New York, NY, USA, 2008. ACM.

[35] R. J. Hulsebosch, Alfons H. Salden, Mortaza S. Bargh, Peter W. G. Ebben, and J. Reitsma. Context sensitive access control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 111–119, New York, NY, USA, 2005. ACM.

[36] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private queries in location based services: Anonymizers are not necessary. In *Proceedings of the 2008 International Conference on Management of Data*, SIGMOD '08, pages 121–132, New York, NY, USA, 2008. ACM.

[37] Bin Yang, Hua Lu, and Christian S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 335–346, New York, NY, USA, 2010. ACM.

[38] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[39] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *SIGOPS Operating Systems Review*, 21(1):8–10, January 1987.

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

[41] Uriel Feige, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 210–217, New York, NY, USA, 1987. ACM.

[42] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '86, pages 186–194, London, UK, 1987. Springer-Verlag.

[43] Federal Financial Institutions Examination Council. *Authentication in an Internet Banking Environment*, October 2005.

[44] Long Nguyen Hoang, Pekka Laitinen, and N. Asokan. Secure roaming with identity metasystems. In *Proceedings of the 7th ACM Symposium on Identity and Trust on the Internet*, IDtrust '08, pages 36–47, New York, NY, USA, 2008. ACM.

[45] Abhilasha Bhargav-Spantzel, Anna C. Squicciarini, and Elisa Bertino. Establishing and protecting digital identity in federation systems. *Journal of Computer Security*, 14(3):269–300, May 2006.

[46] Kyusuk Han and Kwangjo Kim. Enhancing privacy and authentication for location based service using trusted authority. In *2nd Joint Workshop on Information Security*. Information and Communication System Security, 2007.

[47] Jalal Al-Muhtadi, Raqeul Hill, Roy Campbell, and M. Dennis Mickunas. Context and location-aware encryption for pervasive computing environments. In *Proceedings of the 4th Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, PERCOMW '06, pages 289–294, Washington, DC, USA, March 2006. IEEE Computer Society.

[48] Keith Lofstrom, W. Robert Daasch, and Donald Taylor. IC identification circuit using device mismatch. In *IEEE International Solid-State Circuits Conference*, ISSCC '00, pages 372–373, Washington, DC, USA, 2000. IEEE Computer Society.

[49] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 9–14, New York, NY, USA, 2007. ACM.

[50] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical unclonable functions and public-key crypto for FPGA IP protection. In *International Conference on Field Programmable Logic and Applications*, FPL '07, pages 189–195, August 2007.

[51] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 63–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

[53] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 149–160, Washington, DC, USA, 2002. IEEE Computer Society.

[54] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. AEGIS: A single-chip secure processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.

[55] Mikhail J. Atallah, Eric D. Bryant, John T. Korb, and John R. Rice. Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08, pages 45–48, New York, NY, USA, 2008. ACM.

[56] Keith B. Frikken, Marina Blanton, and Mikhail J. Atallah. Robust authentication using physically unclonable functions. In *Proceedings of the 12th International Conference on Information Security*, ISC '09, pages 262–277, Berlin, Heidelberg, September 2009. Springer-Verlag.

[57] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[58] Trusted Computing Group. Trusted Platform Module Main Specification, October 2003. `http://www.trustedcomputinggroup.org/`.

[59] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[60] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1-2):13–22, December 2008.

[61] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 308–317, New York, NY, USA, 2004. ACM.

[62] Srinivas Devadas, G. Edward Suh, Sid Paral, Richard Sowell, Tom Ziola, and Vivek Khandelwal. Design and implementation of PUF-based "unclonable" RFID ICs for anti-counterfeiting and security applications. In *Proceedings of the 2008 IEEE International Conference on RFID*, pages 58–64, Washington, DC, USA, April 2008. IEEE Computer Society.

[63] Boris Danev, Thomas S. Heydt-Benjamin, and S. Čapkun. Physical-layer identification of RFID devices. In *Proceedings of the 18th USENIX Security Symposium*, pages 199–214, Berkeley, CA, USA, 2009. USENIX Association.

[64] Nurbek Saparkhojayev and Dale R. Thompson. Matching electronic fingerprints of RFID tags using the Hotelling's algorithm. In *IEEE Sensors Applications Symposium*, SAS '09, pages 19–24, Washington, DC, USA, February 2009. IEEE Computer Society.

[65] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating System Review*, 42(4):315–328, 2008.

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

[67] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

[68] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

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

[70] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 178–192, New York, NY, USA, 2003. ACM.

[71] Krerk Piromsopa and Richard J. Enbody. Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing*, 3(4):365–376, 2006.

[72] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic software self-healing using rescue points. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 37–48, New York, NY, USA, 2009. ACM.

[73] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies–a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM.

[74] Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh. Using rescue points to navigate software recovery. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 273–280, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, and Angelos D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, ATC '07, pages 17:1–17:14, Berkeley, CA, USA, 2007. USENIX Association.

[76] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATC '05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

[77] Ruirui Huang, Daniel Y. Deng, and G. Edward Suh. Orthrus: Efficient software integrity protection on multi-cores. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 371–384, New York, NY, USA, 2010. ACM.

[78] Oliver Trachsel and Thomas R. Gross. Variant-based competitive parallel execution of sequential programs. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 197–206, New York, NY, USA, 2010. ACM.

[79] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '08, pages 843–848, Washington, DC, USA, 2008. IEEE Computer Society.

[80] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 33–46, New York, NY, USA, 2009. ACM.

[81] Weidong Shi, Hsien-Hsin S. Lee, Laura Falk, and Mrinmoy Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 102–113, Washington, DC, USA, 2006. IEEE Computer Society.

[82] Paul D. Williams and Eugene H. Spafford. CuPIDS: An exploration of highly focused, co-processor-based information system protection. *Computer Networks*, 51(5):1284–1298, 2007.

[83] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation*, OSDI '04, pages 21–34, Berkeley, CA, USA, 2004. USENIX Association.

[84] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pages 2–13, New York, NY, USA, 2008. ACM.

[85] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[87] Armando Fox and David Patterson. Self-repairing computers. *Scientific American*, pages 54–61, June 2003.

[88] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[89] Michael Luby. LT codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 271–280, Washington, DC, USA, 2002. IEEE Computer Society.

[90] Amin Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking*, 14(SI):2551–2567, 2006.

[91] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Berkeley, CA, USA, 2003. USENIX Association.

[92] PaX Team. PaX. `http://pax.grsecurity.net/`.

[93] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[94] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[95] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, Washington, DC, USA, December 2009. IEEE Computer Society.

[96] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[97] Valgrind Team. Valgrind. `http://www.valgrind.org/`.

[98] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, pages 49–54, New York, NY, USA, 2009. ACM.

[99] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[100] Ping Chen, Xiao Xing, Hao Han, Bing Mao, and Li Xie. Efficient detection of the return-oriented programming malicious code. In *Proceedings of the 6th International Conference on Information Systems Security*, ICISS '10, pages 140–155, Berlin, Heidelberg, 2010. Springer-Verlag.

[101] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[102] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th ACM European Conference on Computer Systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[103] Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 New Security Paradigms Workshop*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[104] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28(2):4:1–4:54, July 2010.

[105] Frode Hansen and Vladimir Oleshchuk. Application of role-based access control in wireless healthcare information systems. In *Scandinavian Conference in Health Informatics*, pages 30–33, 2003.

[106] GISToolkit. `http://gistoolkit.sourceforge.net/`.

[107] Open GIS Consortium. Open GIS geography markup language (GML) implementation specification, 2003. `http://www.opengeospatial.org/standards/gml`.

[108] Nokia. Nokia 6131 NFC SDK programmer's guide.

[109] NFC Forum. NFC forum tag type technical specifications. `http://www.nfc-forum.org/`.

[110] Advanced Card Systems Limited. ACR122 NFC contactless smart card reader software development kit. `http://www.acs.com.hk/acr122-sdk.php`.

[111] JSR 257 Expert Group. JSR 257: Contactless communication API.

[112] Francois Kooman. The nfcip-java project. `http://code.google.com/p/nfcip-java/`.

[113] Sreekanth Malladi, Jim Alves-foss, and Robert B. Heckendorn. On preventing replay attacks on security protocols. In *Proceedings of the International Conference on Security and Management*, pages 77–83. CSREA Press, 2002.

[114] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, pages 255–268, Washington, DC, USA, 2000. IEEE Computer Society.

[115] Paul Syverson. A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, CSFW '94, pages 187–191, Washington, DC, USA, June 1994. IEEE Computer Society.

[116] S. Gritzalis, D. Spinellis, and Sena Sa. Cryptographic protocols over open distributed systems: A taxonomy of flaws and related protocol analysis tools. In *16th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '97, pages 123–137, September 1997.

[117] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[118] Christian S. Jensen, Hua Lu, and Bin Yang. Indoor–a new data management frontier. *IEEE Data Engineering Bulletin*, 33(2):12–17, June 2010.

[119] Christian S. Jensen, Hua Lu, and Bin Yang. Graph model based indoor tracking. In *Proceedings of the 10th International Conference on Mobile Data Management*, MDM '09, pages 122–131, Washington, DC, USA, 2009. IEEE Computer Society.

[120] Christian Becker and Frank Dürr. On location models for ubiquitous computing. *Personal Ubiquitous Computing*, 9(1):20–31, January 2005.

[121] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140, London, UK, 1992. Springer-Verlag.

[122] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[123] Amit Sahai and Brent Waters. Fuzzy identity based encryption. In *Proceedings of the 24th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '05, pages 457–473, Berlin, Heidelberg, 2005. Springer-Verlag.

[124] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of the 36th Symposium on Foundations of Computer Science*, FOCS '95, pages 41–50, Washington, DC, USA, 1995. IEEE Computer Society.

[125] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[126] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Jean-François Raymond. Breaking the O(n1/(2k-1)) barrier for information-theoretic private information retrieval. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 261–270, Washington, DC, USA, 2002. IEEE Computer Society.

[127] Sergey Yekhanin. *Locally decodable codes and private information retrieval schemes*. PhD thesis, MIT, 2007.

[128] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 364–373, Washington, DC, USA, 1997. IEEE Computer Society.

[129] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '99, pages 402–414, Berlin, Heidelberg, 1999. Springer-Verlag.

[130] Helger Lipmaa. An oblivious transfer protocol with log-squared total communication. In *Proceedings of the 8th International Conference on Information Security*, ISC '05, pages 314–328, Berlin, Heidelberg, 2005. Springer-Verlag.

[131] Yan-Cheng Chan. Single database private information retrieval with logarithmic communication. In *Proceedings of Australasian Conference on Information Security and Privacy*, ACISP, pages 50–61, Berlin, Heidelberg, 2004. Springer-Verlag.

[132] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[133] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, pages 803–815. Springer-Verlag, 2005.

[134] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[135] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, STOC '99, pages 245–254, New York, NY, USA, 1999. ACM.

[136] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[137] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. pages 10–18, 1984.

[138] Jan Camenisch, Maria Dubovitskaya, and Gregory Neven. Oblivious transfer with access control. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 131–140, New York, NY, USA, 2009. ACM.

[139] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2, 1983.

[140] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, April 2007.

[141] Anusha Iyer and Hung Q. Ngo. Towards a theory of insider threat assessment. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 108–117, Washington, DC, USA, 2005. IEEE Computer Society.

[142] INFOSEC Research Council (IRC). Hard problem list, 2005.

[143] CSO Magazine and CERT and United States Secret Service. 2004 e-crime watch survey: Summary of findings. `http://www.cert.org/archive/pdf/2004eCrimeWatchSummary.pdf`, 2004.

[144] Insider threat study: Illicit cyber activity in the banking and finance sector. `http://www.secretservice.gov/ntac/its_report_040820.pdf`, August 2004.

[145] Frank L. Greitzer, Andrew P. Moore, Dawn M. Cappelli, Dee H. Andrews, Lynn A. Carroll, and Thomas D. Hull. Combating the insider cyber threat. *IEEE Security and Privacy*, 6(1):61–64, January 2008.

[146] Joel Predd, Shari Lawrence Pfleeger, Jeffrey Hunker, and Carla Bulford. Insiders behaving badly. *IEEE Security and Privacy*, 6(4):66–70, July 2008.

[147] Verizon RISK Team. 2010 data breach investigations report. Technical report, 2010.

[148] Ellen Messmer. Black hat: Researcher claims hack of chip used to secure computers, smartcards. *Computer World*, February 2010. http://www.computerworld.com/s/article/9151158/Black_Hat_Researcher_claims_hack_of_chip_used_to_secure_computers_smartcards?source=CTWNLE_nlt_security_2010-02-03.

[149] Michael S. Kirkpatrick and Elisa Bertino. Physically restricted authentication with trusted hardware. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, pages 55–60, New York, NY, USA, 2009. ACM.

[150] Michael S. Kirkpatrick and Sam Kerr. Enforcing physically restricted access control for remote data. In *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 203–212, New York, NY, USA, 2011. ACM.

[151] Feng Hao, Ross Anderson, and John Daugman. Combining crypto with biometrics effectively. *IEEE Transactions on Computers*, 55(9):1081–1088, 2006.

[152] Martyn Riley and Iain Richardson. Reed-solomon codes. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html, 1998.

[153] Christophe Devine. FIPS-180-1 compliant SHA-1 implementation. http://csourcesearch.net/c/fid1A3BFA49A2F9E1FFB3147B7238E287C22E7ED0A3.aspx, 2006.

[154] Simon Rockliff. The error correcting codes (ecc) page. http://www.eccpage.com/, 2008.

[155] Michael O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, MIT Laboratory for Computer Science, January 1979.

[156] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *Proceedings of the 28th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '08, pages 39–56, Berlin, Heidelberg, 2008. Springer-Verlag.

[157] Ironkey military strength flash drives. http://www.ironkey.com/, 2010.

[158] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proceedings of the 18th USENIX Security Symposium*, pages 299–315, Berkeley, CA, USA, 2009. USENIX Association.

[159] Srivatsan Narayanan, Ananth Raghunathan, and Ramarathnam Venkatesan. Obfuscating straight line arithmetic programs. In *Proceedings of the 9th ACM Workshop on Digital Rights Management*, DRM '09, pages 47–58, New York, NY, USA, 2009. ACM.

[160] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing*, STC '06, pages 27–42, New York, NY, USA, 2006. ACM.

[161] Vladimir Kolesnikov. Truly efficient string oblivious transfer using resettable tamper-proof tokens. In Daniele Micciancio, editor, *Proceedings of the 7th Theory of Cryptography Conference*, volume 5978, pages 327–342, Berlin, Heidelberg, 2010. Springer Berlin.

[162] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, *Proceedings of the 7th Theory of Cryptography Conference*, volume 5978 of *TCC*, pages 308–326, Berlin, Heidelberg, 2010. Springer Berlin.

[163] Eric Brier, Benoît Chevallier-mames, Mathieu Ciet, Christophe Clavier, and École Normale Supérieure. Why one should also secure RSA public key elements. In *Proceedings of the 8th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 4249 of *CHES '06*, pages 324–338, Berlin, Heidelberg, 2006. Springer-Verlag.

[164] Alexandre Berzati, Cécile Canovas, and Louis Goubin. Perturbating RSA public keys: An improved attack. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 5154 of *CHES '08*, pages 380–395, Berlin, Heidelberg, 2008. Springer-Verlag.

[165] Alexandre Berzati, Cécile Canovas, and Louis Goubin. In(security) against fault injection attacks for CRT-RSA implementations. *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 101–107, 2008.

[166] Alexandre Berzati, Cécile Canovas, Jean-Guillaume Dumas, and Louis Goubin. Fault attacks on RSA public keys: Left-to-right implementations are also vulnerable. In *Proceedings of the Cryptographers' Track at the RSA Conference 2009 on Topics in Cryptology*, CT-RSA '09, pages 414–428, Berlin, Heidelberg, 2009. Springer-Verlag.

[167] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-based attack of RSA authentication. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 855–860, 3001 Leuven, Belgium, Belgium, March 2010. European Design and Automation Association.

[168] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *Proceedings of the 6th Theory of Cryptography Conference*, TCC '09, pages 474–495, Berlin, Heidelberg, 2009. Springer-Verlag.

[169] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, 2001. Springer-Verlag.

[170] Michael S. Kirkpatrick, Sam Kerr, and Elisa Bertino. PUF ROKs: Generating read-once keys with physically unclonable functions (extended abstract). In *6th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, April 2010.

[171] Michael S. Kirkpatrick, Sam Kerr, and Elisa Bertino. PUF ROKs: A hardware approach to read-once keys. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 155–164, New York, NY, USA, 2011. ACM.

[172] KNJN FPGA development boards. `http://www.knjn.com/FPGA-FX2.html`, 2010.

[173] Encryption for ARM MCUs. `http://ics.nxp.com/literature/presentations/microcontrollers/pdf/nxp.security.innovation.encryption.pdf`, 2010.

[174] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[175] PolarSSL: Small cryptographic library. `http://www.polarssl.org/`, 2008.

[176] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to differential power analysis and related attacks. Technical report, Cryptography Research, 1998.

[177] Vijay Sundaresan, Srividhya Rammohan, and Ranga Vemuri. Defense against side-channel power analysis attacks on microelectronic systems. In *IEEE National Aerospace and Electronics Conference*, NAECON '08, pages 144–150, Washington, DC, USA, July 2008. IEEE Computer Society.

[178] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[179] F-Secure. Stuxnet questions and answers. `http://www.f-secure.com/weblog/archives/00002040.html`, 2010.

[180] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Surveys*, 34(3):375–408, September 2002.

[181] David J. C. Mackay. Fountain codes. *IEE Communications*, 152(6):1062–1068, 2005.

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

[183] Zhi Wang and Xuxian Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[184] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATC '05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[185] Frank Uyeda, Huaxia Xia, and Andrew A. Chien. Evaluation of a high performance erasure code implementation. Technical Report CS2004-0798, UCSD Computer Science and Engineering, September 2004.

[186] Uwe F. Mayer. Linux/Unix nbench. `http://www.tux.org/~mayer/linux/bmark.html`, 2003.

[187] Eugene H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University, West Lafayette, IN, USA, 1988.

[188] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996.

[189] Scut / Team Teso. Exploiting format string vulnerabilities, 2001.

[190] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[191] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Berkeley, CA, USA, 1998. USENIX Association.

[192] Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59(9), June 2002.

[193] Solar Designer. Getting around non-executable stack (and fix). August 1997.

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

[195] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

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

[197] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

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

[199] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC '10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[200] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies*, WOOT '10, pages 1–10, Berkeley, CA, USA, 2010. USENIX Association.

[201] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. Return-oriented rootkit without returns (on the x86). In *Proceedings of the 12th International Conference on Information and Communications Security*, ICICS '10, pages 340–354, Berlin, Heidelberg, 2010. Springer-Verlag.

[202] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Jump-oriented programming: A new class of code-reuse attack. Technical Report TR-2010-8, North Carolina State University, 2010.

[203] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-independent programs. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 547–558, New York, NY, USA, 2010. ACM.

VITA

VITA

Education

Purdue University, West Lafayette, IN

Ph.D., Computer Science 08/2011

Adviser: *Elisa Bertino*

Dissertation: "Trusted Enforcement of Contextual Access Control"

Michigan State University, East Lansing, MI

M.S., Computer Science & Engineering 08/2007

Adviser: *Richard Enbody*

Thesis: "Canary Bit: Extending Secure Bit for Data Pointer Protection from Buffer Overflow Attacks"

Indiana University, Bloomington, IN

B.A., Mathematics & Computer Science 05/2001

Minor: Religious Studies

Research Interests

Information security, access control, applied cryptography, embedded systems

Research Experience

Purdue University West Lafayette, IN

Research Assistant 08/2009 to present

Supervised by *Dr. Elisa Bertino*

Conducted independent research on contextual access control systems and trusted platform technologies, utilizing microkernel OS, cell phones enabled with Bluetooth and near-field communication (NFC) technologies, physically unclonable functions (PUFs), field-programmable gate arrays (FPGAs), virtual machine monitors. Co-authored multiple peer-reviewed papers, position papers, and a textbook chapter. Co-authored a successful partnership proposal with Sypris Solutions, Inc., and contributed to multiple grant proposals. Coordinated weekly research group meetings.

Sypris Electronics Tampa, FL
Cybersecurity Research Intern 05/2010 to 08/2010
Supervised by *Dr. Hal Aldridge*

Conducted research on architectural and OS approaches for trustworthy computing. Proposed a novel hardware architecture (now under submission for a patent) for secure computing. Initiated research on customized execution environments for application isolation.

Oak Ridge National Laboratory Oak Ridge, TN
HERE@ORNL Summer Intern 05/2009 to 08/2009
Supervised by *Dr. Frederick Sheldon*

Conducted research on use of PUFs for cyber-physical systems, such as smart grid systems. Developed prototype for contextual insider threat detection for Linux kernel.

Grant Experience

Sypris Electronics January 2010

Co-authored a successful partnership proposal for studying techniques to protect remote, unattended devices. Award of $114,820 under proposal COEUS 10086817. Funding supported three graduate students for one year.

Publications

Peer-reviewed Conference & Workshop Papers

1. ***Michael S. Kirkpatrick***, Sam Kerr, and Elisa Bertino, "PUF ROKs : A Hardware Approach to Read-Once Keys." *6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 10 pages, Hong Kong, March 2011.

2. ***Michael S. Kirkpatrick*** and Sam Kerr, "Enforcing Physically Restricted Access Control for Remote Data." *1st ACM Conference on Data and Application Security and Privacy (CODASPY)*, 11 pages, San Antonio, TX, February 2011.

3. Sam Kerr, ***Michael S. Kirkpatrick***, and Elisa Bertino, "PEAR: A Hardware-based Authentication System." *3rd ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS (SPRINGL)*, 10 pages, San Jose, California, November 2010.

4. ***Michael S. Kirkpatrick*** and Elisa Bertino, "Enforcing Spatial Constraints for Mobile RBAC Systems." *15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 10 pages, Pittsburgh, PA, June 2010.

5. ***Michael S. Kirkpatrick*** and Elisa Bertino, "Software Techniques to Combat Drift in PUF-based Authentication Systems." *Secure Component and System Identification (SECSI)*, 9 pages, Cologne, Germany, April 2010.

6. ***Michael S. Kirkpatrick***, Sam Kerr, and Elisa Bertino, "PUF ROKs: Generating Read-Once Keys with Physically Unclonable Functions." Extended abstract, *6th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, 4 pages, Oak Ridge, TN, April 2010.

7. ***Michael S. Kirkpatrick*** and Elisa Bertino, "Physically Restricted Authentication with Trusted Hardware." *Fourth Annual Workshop on Scalable Trusted Computing*, 6 pages, Chicago, IL, November 2009.

8. ***Michael S. Kirkpatrick*** and Elisa Bertino, “Context-Dependent Authentication and Access Control.” *Open Research Problems in Network Security (iNetSec)*, 13 pages, Zurich, Switzer- land, April 2009.

Book Chapters

9. Elisa Bertino, Stephen J. Elliott, ***Michael S. Kirkpatrick***, and Shimon K. Modi, “Digital Identity Management.” *Security in Computing and Networking Systems: The State-of-the-Art*, edited by W. McQuay and W. W. Smari, 2010 (*to appear*).

Invited Papers

10. Aditi Gupta, Salmin Sultana, ***Michael S. Kirkpatrick***, and Elisa Bertino, “A Selective Encryption Approach to Fine-Grained Access Control for P2P File Sharing.” *The 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 10 pages, Chicago, IL, October 2010.

11. Elisa Bertino and ***Michael S. Kirkpatrick***, “Location-Aware Authentication and Access Control – Concepts and Issues.” *IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA)*, 6 pages, Bradford, UK, May 2009.

Position Papers

12. Carmen R. Vicente, ***Michael S. Kirkpatrick***, Gabriel Ghinita, Elisa Bertino, and Christian S. Jensen, “Requirements and Challenges of Location-Based Access Control in Healthcare Emergency Response.” Position Paper, *2nd SIGSPATIAL ACM GIS International Workshop on Security and Privacy in GIS and LBS (SPRINGL)*, 6 pages, Seattle, WA, November 2009.

13. ***Michael S. Kirkpatrick***, Elisa Bertino, and Frederick T. Sheldon, "Restricted Authentication and Access Control for Cyber-physical Systems." *DHS Workshop on Future Directions in Cyber-physical Systems Security*, 5 pages, Newark, NJ, July 2009.

Works Under Submission

14. ***Michael S. Kirkpatrick***, Gabriel Ghinita, and Elisa Bertino, "Resilient Authenticated Execution of Critical Applications in Untrusted Environments," 14 pages.
15. ***Michael S. Kirkpatrick***, Gabriel Ghinita, and Elisa Bertino, "Privacy-preserving Enforcement of Spatially Aware RBAC," 14 pages.
16. ***Michael S. Kirkpatrick***, Sam Kerr, and Elisa Bertino, "Marlin: Minimization and Randomization as Defense against Library Attacks," 11 pages.

Patents

William P. Izzo, Hal A. Aldridge, Anthony G. Frazier, and ***Michael S. Kirkpatrick***, "Discretely Allocated Multicore Processing System." Patent application submitted by Sypris Electronics, Inc. October 2010.

Teaching Interests

Security & cryptography, operating systems, computer organization & architecture, discrete mathematics, data structures, embedded systems, C/C++, Perl

Teaching Experience

Purdue University, Department of Computer Science West Lafayette, IN

- *Lecturer*, Foundations of Computer Science (CS 18200) 08/2010 – 12/2010
  Presented lecture material for one 60-student section as an independent instructor. Developed course material and lecture guides. Prepared daily lesson plans. Co-authored midterm and final exams.

- *Research Co-adviser*, Sam Kerr 09/2009 – *present*

  Supervised undergraduate student through multiple projects, involving implementation work for PUF designs in FPGAs and microkernel OS development. His work was used in multiple peer-reviewed papers.

- *Research Co-adviser*, Shuai Liu 09/2010 – 12/2010

  Supervised undergraduate student for a project that involved using Bluetooth to exchange data between an Android cell phone and a computer.

Purdue University, Mathematics West Lafayette, IN

- *Instructor*, College Algebra (MA 15200) 01/2009 to 05/2009

  Taught two sections of 35-40 students each as an independent instructor. Prepared lesson plans and daily quizzes. Graded quizzes and homework.

- *Recitation Instructor*, Precalculus (MA 15900) 08/2008 to 12/2008

  Led two recitation sections of 35-40 students each, reinforcing primary lectures with additional examples and answering questions. Prepared lesson plans and graded daily quizzes.

Purdue University, Department of Computer Science West Lafayette, IN

- *Lab Instructor*, Foundations of Computer Science (CS 182) 01/2008 to 05/2008

  Led one discussion section of 35 students, reinforcing primary lectures with additional examples and answering questions. Graded bi-weekly homework assignments and exams.

- *Teaching Assistant*, Engineering Projects in Community Service (EPICS) 08/2007 to 12/2007

  Served as technical and managerial reference for three project teams, ranging from 4 to 25 students. Graded design notebooks, written assignments, and presentations.

- *Lab Instructor*, Data Structures (CS 251) 01/2007 to 05/2007

  Led one lab section of 20 students, explaining programming assignment requirements and answering related questions. Graded projects and exams.

Teaching Certifications & Training

Center for Instructional Excellence (CIE) West Lafayette, IN

Graduate Teacher Certification (GTC) May 2010

Requirements: Two semesters teaching experience, attendance at GTA orientation (including microteaching), six hours of instructional improvement, classroom videotaping & review, and a written self-reflection based on student evaluation.

College Teaching Workshops: Teaching Principles and Techniques, Creating an Optimal Learning Environment, Teaching Effective Labs, Policies & Procedures, Student-Teacher Relationships, Designing Instruction, Presentation Techniques to Enhance Learning, Using Feedback & Assessment to Improve Your Teaching, Discussion Techniques to Enhance Learning, Using Objective Tests, Using Subjective Tests and Assigning Grades, Dealing with Cheating: Prevention & Response

Graduate Coursework

Purdue CS Courses: Compilers, Operating Systems, Information Security, Cryptography, Programming Languages, Algorithms, Database Security, Advanced Information Assurance, Embedded Systems (audit), Probability Theory

Michigan State CSE Courses: Advanced Operating Systems, Formal Methods in Software Development, Computer & Network Security, Artificial Intelligence

Activities & Honors

ACM Computer & Communication Security Conference

- Student Travel Grant, Recipient 11/2009
- Workshop Travel Grant, Recipient 11/2009

Upsilon Pi Epsilon International CS Honor Society 04/2009 – *present*

Golden Key International Honour Society 03/2008 – *present*

Alpha Phi Omega, Mu Chapter (Indiana University)

- Distinguished Service Key (DSK) 11/2001

American Mensa 07/2000

Professional Activities

ACM Student Member 03/2009 – *present*

- Member of Special Interest Group on Security, Audit, and Control (SIGSAC)
- Member of Special Interest Group on Computer Science Education (SIGCSE)

Purdue CS Graduate Student Board Chair 08/2009 – 08/2010

Conference Program Committee

- 6th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW), 2010

Conference & Journal Reviewer

- ACM Conference on Data and Application Security and Privacy (CODASPY)
- ACM Symposium on Access Control Models and Technologies (SACMAT)
- Annual International Conference on Financial Cryptography and Data Security (FC)
- Annual Computer Security Applications Conference (ACSAC)
- Annual Cyber Security and Information Intelligence Research Workshop (CSI-IRW)
- Hawai'i International Conference on System Sciences
- IEEE Transactions on Dependable and Secure Computing
- IEEE Transactions on Information Forensics & Security
- IEEE Transactions on Very Large Scale Integration Systems
- IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)

- IEEE International Conference on Data Mining
- International Conference on Mobile Data Management
- International Journal of Information Security
- "Security in Computing," special issue of Transactions on Computational Science, 2008

Other Work Experience

IBM Server & Technology Group *Software Engineer*

Essex Junction, VT 06/2002 to 12/2006

Developed and implemented Optical Proximity Correction (OPC) solutions using the SVRF and DFFL shapes processing engines. Designed, built, maintained and debugged infrastructure programs written in both Perl and Korn shell. Performed failure analysis to support product engineering groups. Created the security foundation for a CGI-based database application that leveraged LDAP authentication and AFS permissions. Created multiple clones of the Data Prep system for use by separate design automation clients. Developed and implemented a mechanism that granted corporate alliance partners controlled access to proprietary data.

IBM Global Services *IT Professional*

Chicago, IL 06/2001 to 05/2002

Provided consulting services in a professional, client-driven environment. Performed testing of web development projects in the education and finance sectors. Developed HTML and JavaScript front-ends as part of a creative design team, with clients in the education, finance, and retail sectors. Created presentations for internal customers and executives.

CourseShare.com *Programmer*

Bloomington, IN 10/2000 to 05/2001

Designed and implemented a PHP web application for online survey creation. Built the system to interact with an Access database and to use JavaScript for Dynamic HTML. Served as technical lead for the student team. Responsible for training developers to use PHP.

Indiana University *Programmer*

Bloomington, IN 07/2000 to 05/2001

Designed and implemented a ColdFusion web application for tracking personnel information of the student consulting division. Built the system to communicate with a SQL Server back-end, using JavaScript for the Dynamic HTML front-end.