Fall 2014

Secure platforms for enforcing contextual access control

Aditi Gupta
Purdue University

Graduate School Form
(Revised /)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By ADITI GUPTA

Entitled

SECURE PLATFORMS FOR ENFORCING CONTEXTUAL ACCESS CONTROL

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

ELISA BERTINO

MIKHAIL J. ATALLAH

NINGHUI LI

SONIA FAHMY

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

ELISA BERTINO

Approved by Major Professor(s): _____

_____

Approved by: SUNIL PRABHAKAR / WILLIAM J. GORMAN 09/26/2014

Head of the Graduate Program Date

SECURE PLATFORMS FOR ENFORCING CONTEXTUAL ACCESS CONTROL

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Aditi Gupta

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2014

Purdue University

West Lafayette, Indiana

To my loving parents,
Anjali Gupta and Arvind Gupta,
for giving me the courage to pursue my dreams.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere thanks and gratitude to my advisor, Prof. Elisa Bertino for her invaluable guidance while supervising this thesis. Her extraordinary patience, support and encouragement were crucial in completion of this dissertation. I am grateful to Prof. Mikhail J. Atallah, Prof. Ninghui Li and Prof. Sonia Fahmy for agreeing to be on my PhD committee and providing invaluable feedback on my doctoral work.

I would like to extend my special thanks to Dr. N. Asokan for mentoring me during my two internships at Nokia Research and introducing me to the area of contextual security. I thoroughly enjoyed working on the contextual security project which eventually became a part of this dissertation. I would also like to thank Markus Miettinen for several interesting discussions during the internship.

I would like to thank all my colleagues and collaborators at Purdue for numerous discussions and research insights. Special thanks to Michael S. Kirkpatrick and Javid Habibi whom I closely collaborated with on several research projects. I would like to thank CS graduate office staff for their help with various administrative tasks, with special thanks to William J. Gorman for his invaluable advice on several matters.

I would also like to thank my friends who made my stay at Purdue a truly enjoyable experience. I am deeply grateful to my loving parents, Anjali Gupta and Arvind Gupta, for their unconditional love, support and encouragement. I would like to thank them for always believing in me and for providing me with the best opportunities while growing up. I would like to thank my brother, Ankur Gupta, who always knows how to cheer me up and made me see the bright side when life got tough. Last but not the least, I would like to thank my loving husband, Utsaw Kumar. This thesis would not have been possible without his encouragement and extraordinary support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Gupta, Aditi Ph.D., Purdue University, December 2014. Secure Platforms for Enforcing Contextual Access Control. Major Professor: Elisa Bertino.

Advances in technology and wide scale deployment of networking enabled portable devices such as smartphones has made it possible to provide pervasive access to sensitive data to authorized individuals from any location. While this has certainly made data more accessible, it has also increased the risk of data theft as the data may be accessed from potentially unsafe locations in the presence of untrusted parties. The smartphones come with various embedded sensors that can provide rich contextual information such as sensing the presence of other users in a context. Frequent context profiling can also allow a mobile device to learn its surroundings and infer the familiarity and safety of a context. This can be used to further strengthen the access control policies enforced on a mobile device. Incorporating contextual factors into access control decisions requires that one must be able to trust the information provided by these context sensors. This requires that the underlying operating system and hardware be well protected against attacks from malicious adversaries.

In this work, we explore how contextual factors can be leveraged to infer the safety of a context. We use a context profiling technique to gradually learn a context's profile, infer its familiarity and safety and then use this information in the enforcement of contextual access policies. While intuitive security configurations may be suitable for non-critical applications, other security-critical applications require a more rigorous definition and enforcement of contextual policies. We thus propose a formal model for proximity that allows one to define whether two users are in proximity in a given context and then extend the traditional RBAC model by incorporating these proximity constraints. Trusted enforcement of contextual access control requires that the

underlying platform be secured against various attacks such as code reuse attacks. To mitigate these attacks, we propose a binary diversification approach that randomizes the target executable with every run. We also propose a defense framework based on control flow analysis that detects, diagnoses and responds to code reuse attacks in real time.

1 INTRODUCTION

Mobile devices such as smartphones and tablets are fast becoming an integral part of life for many users. They are used for performing everyday tasks like Email and Internet banking that involves storing sensitive data on the device. They also contain personal data like photos and videos, communication logs, location information and logs of monetary transactions. Earlier mobiles devices were simple feature phones with basic functionality of making phone calls. Now these devices have gradually evolved into powerful computational devices with embedded sensors and network connectivity. Users can now access and process data from anywhere using their smartphones or tablets. However, pervasive access to data implies that a user can also access data from potentially unsafe contexts. This can lead to leakage of sensitive information. For instance, an employee can access his corporate email from his mobile device while sitting in a cafe. A malicious adversary or even a curious bystander may gain access to confidential corporate information by shoulder-surfing attack. Further, if the user's device is connected to an untrusted or unsecured access point such as free WiFi at the cafe, then a malicious adversary can steal user's information such as login and passwords by eavesdropping on the unencrypted traffic. Thus, access control policies that assume static unchanging context are not sufficient for mobile devices and must incorporate contextual factors while evaluating access request.

Contextual access control incorporates various contextual factors such as location of the user while specifying and enforcing access policies. For example, a policy based on location may allow a doctor to access patients' medical records only when he or she is physically present at the hospital. Here, the location of the user determines the context of the user. However, the notion of context is quite vague and application dependent. In the realm of context-aware computing, Dey [1] defines context as *"any information that can be used to characterize the situation of an entity. An entity is*

a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." A context may comprise of one or more context variables such as location of the user, other users in proximity, time of the day, temperature, noise levels etc. Different applications may consider a different set of contextual factors to identify a context. For example, applications such as friend finder and deal finder consider only the location of a user as a contextual factor while other applications such as mobile apps for public transport schedule may consider both location and time.

Presence of other users in proximity is another contextual factor that can affect the safety of a context for access control applications. For example, in a military setting, it may be unsafe to access a top secret document in the presence of any civilian. Another example, would be when presence of other users, say a supervisor, makes the context safe for certain actions. For instance, a nurse may be allowed to access a patient's medical data only in the presence of a doctor. That is, proximity based access control can be considered as an instance of contextual access control where access policies allow or deny access requests based on the presence of *other* users in proximity. Specifying access policies in such settings requires a precise and formal definition of proximity. Absence of a formal model for proximity can lead to ambiguous interpretations while enforcing access control.

A user's context is inherently dynamic in nature due to the constantly changing context variables and this may affect the safety of the context. If sensitive data is accessed from an unsafe context, then the security of this data may be compromised. This calls for strong protection mechanisms on mobile devices. Protection mechanisms serve their purpose only when they are configured with sensible policies for accessing and sharing data. However, managing a large number of policy configurations can be quite overwhelming and unintuitive for a user. Application and service designers attempt to tackle the usability problem by providing users with a default policy configuration. But a global default policy may not be suitable for the needs of every user. Users are therefore left with two unsatisfactory alternatives: either use

one-size-fits-all default policies which may not be sensible, or, suffer through manually configuring the bulk of policies by hand which may not be intuitive or easy-to-use.

Modern smartphones are equipped with a variety of sensors capable of continuously monitoring a wide range of parameters such as location, Bluetooth and WiFi devices in the neighborhood, temperature, ambient light, noise levels etc. These observations characterize the *context* of a device, and hence of its user. We argue that by *profiling contexts* in terms of how the context parameters change over time, we can infer appropriate access and sharing policies for sensitive data on the device, which can help towards at least partially automating the process of setting sensible policies.

Even if the access request is made from a safe context, no guarantees on correct enforcement can be provided if the device itself has been compromised. This risk is elevated by the *bring you own device* (BYOD) trend where corporate organizations allow employees to bring their personal devices to work and connect it to the corporate network. This puts the security of corporate network at a risk since a malware infected device can infect other devices on the network and steal confidential information. Correct enforcement of contextual access control requires that the information received from contextual sensors is correct and the system has not been compromised. A malicious adversary may compromise the system by exploiting some vulnerability such as buffer overflow and then hijacking the control flow. Return oriented programming (ROP) attack is one such attack that can execute arbitrary logic by reusing the 'good' code in the victim system towards malicious purposes. It is necessary to build tools and techniques that that will protect systems against these kinds of attacks.

This work examines the feasibility of developing secure platforms for enforcing contextual access control. In particular, it discusses solutions to three problems. First, how can a mobile device automatically infer the safety of a context and intuitively configure access policies. Second, how to formally define proximity and incorporate proximity constraints in contextual access control. Third, how to secure the underlying platform against code-reuse attacks so that access control can be correctly enforced. These are discussed in more details below.

1.1 Intuitive Security Policy Configuration

Configuring access control policies in mobile devices can be quite tedious and unintuitive for users. Software designers attempt to address this problem by setting up default policy configurations. But such global defaults may not be sensible for all users. We conjecture that profiling a variety of contextual information can be used to infer the familiarity and safety of a context and aid in access control decisions.

As an illustrative example, consider the case of *device locks*: Mobile devices have a device lock feature similar to the screen-saver lock on PCs. When the device has been idle for a pre-defined fixed period of time, the device lock kicks in. Thereafter the user has to unlock the device before accessing the applications and data on the device. A device may support multiple unlocking methods like a slider or passcode entry but a specific unlocking method has to be chosen when the device lock feature is enabled. In an enterprise, the enterprise system administrator may force its users to use strong device lock if the device is capable of accessing enterprise data like corporate e-mail or intranet websites. Suppose a user, Alice, finds it very inconvenient having to type in a passcode several times every day. She may decide to disable the device lock and risk the compromise of her sensitive data like e-mails, or she may opt to remove applications like corporate e-mail that mandate the use of device lock.

Alice's experience with device lock can be significantly improved by making the device lock to adapt its behavior based on the context. Instead of having a fixed pre-defined timeout for the device lock to kick-in and always using the same unlocking method, the device lock application could use dynamic configuration of these parameters depending on the device context. For example, in a safe and familiar place like her home where the likelihood of the device being stolen is low, Alice would like to have a long timeout, and a "shallow" unlocking method like a slider (that does not tax her too much), whereas in an unfamiliar place she would be willing to live with a very short timeout and a "deeper" unlocking method like passcode entry.

The question then is "how can the device estimate the familiarity and safety for a context at any given time?" We propose a framework to estimate the familiarity and safety of a context at any instant and use these values to dynamically configure security policies. In this approach, the device periodically scans its environment for a variety of context variables like GPS readings, WiFi access points and Bluetooth devices. Based on these scans, the device discovers contexts that are encountered repeatedly; these are likely personal *contexts of interest (CoIs)* for the user. The device then profiles these CoIs by keeping track of which WiFi and Bluetooth devices are encountered in a given CoI and the nature of those encounters. These profiles can be used to estimate the *familiarity* of a device with respect to a context. The inferred device familiarity values can then be used to estimate the familiarity of a context itself. The device then uses current and historically aggregated context familiarity information to estimate the safety of the current context.

This basic approach needs to be complemented by allowing the user to provide feedback about the perceived safety of a context. Feedback is important in two respects: either the user wants to speed up the learning process or wants to correct incorrectly inferred estimated safety of a context.

User privacy is an explicit design principle for our framework. We do not want either the raw sensor data or the inferred contextual parameters to leave the user's device. Therefore, all of the contextual data is stored and processed locally on the user's device itself which enables better security and privacy protection.

Intuitive policy configuration may not be suitable for applications that require strong security guarantees. For example, security applications in military setting may require explicit policy configuration and precise definition of the context variables of interest. In the next subsection, we discuss formal model for one such context variable, that is proximity.

1.2 Formal Proximity Model for Access Control

To combat the threat of information leakage through pervasive access, researchers have proposed several extensions to the popular role-based access control (RBAC) model. Such extensions can incorporate contextual features, such as location, into the policy decision in an attempt to restrict access to trustworthy settings. In many cases, though, such extensions fail to reflect the true threat, which is the presence or absence of *other users*, rather than absolute locations. For instance, for location-aware separation of duty, it is more important to ensure that two people are in the same room, rather than in a designated, pre-defined location.

Prox-RBAC [2] was proposed as an extension to consider the relative *proximity* of other users with the help of a pervasive monitoring infrastructure. However, we have identified two shortcomings with Prox-RBAC as previously proposed. First, the model relies on an intuitive notion that "proximity" means the users are present (or not) within the same physical space. This lack of a rigorous understanding of proximity can lead to surprising interpretations. For instance, in Prox-RBAC, two users at opposite ends of a building could be considered to be within proximity for one policy; however, for another policy, two users standing in adjacent rooms on opposite sides of the same door would *not* be in proximity of one another. As such, this informal approach allows for entities to be in proximity, despite the fact that they are not physically close.

Second, we find the exclusive focus on the spatial domain to be unnecessarily restrictive. The intuition that proximity indicates relative closeness of two entities can be applied in several domains with interesting results. For instance, a temporal proximity constraint could require that two people digitally sign a document within 24 hours. In attribute-based proximity, an Assistant Professor and an Associate Professor have professions (*i.e.,* attributes) that are similar. Clearly, a unified and formal definition of proximity can be applied to a wide variety of settings.

We have analyzed five contextual domains, or *realms*, namely geographic, attribute-based, social, cyber, and temporal realms for defining proximity. In this work, we define these realms and show how they can be mapped onto a unified abstract space model. We then apply the calculus-based method [3] for defining topological relations on *features* in order to specify a formal distance metric. We use this metric to define two forms of proximity, specifically *weak role proximity* and *strong role proximity*. In both forms, proximity specifies that two entities must have a distance measure (in the abstract space) that is less than some threshold value.

It is important to emphasize the advantages of this formal approach. First, by grounding the notion of proximity in terms of a distance and threshold values, we ensure that our formalisms reflect the intuition of proximity as closeness. As such, the mandatory specification of a metric reduces the likelihood of surprising interpretations of proximity. Second, by defining proximity in terms of an abstract space model, our approach is very flexible and simplifies the adaptation of policies for other realms beyond the five we consider. That is, mapping the realms onto the abstract space model allows us to define a common framework for enforcing the policy constraints; adapting the model and policies for additional realms would only require mapping the realm onto the abstract space model. Finally, by defining a common enforcement architecture, it is possible to develop reusable code libraries and protocols that could be applied to any enforcement architecture that maps onto our abstract space model.

1.3 Securing Systems against Code Reuse Attacks

Correct enforcement of contextual access control requires that the underlying platforms are secure and able to defend against attacks from malicious adversaries. As a first step, we explore the defense techniques for conventional systems such as PCs. These techniques can then be applied to other non conventional systems such as mobile devices and other embedded systems. In particular, we focus on code-reuse attacks, including return-oriented programming (ROP) and jump-oriented program-

ming (JOP), that bypass defenses against code injection by repurposing existing executable code toward a malicious end.

A common feature of these attacks is the reliance on the knowledge of the layout of the executable code. Early solutions to the problem of code-reuse 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, a single exploit packet would not be effective on all running instances of an application. Although randomization initially seemed promising, these solutions suffered from the small amount of randomization possible [4]. Consequently, successful brute-force attacks were feasible. We propose to re-examine the granularity at which randomization is performed as a defense against ROP attacks.

Our system, *Marlin*, introduces a randomization technique that shuffles the code blocks in an application binary. This technique is integrated into a customized bash shell that randomizes the target binary at load time just before execution. This randomization approach has many benefits. First, for any decent-sized code base with a large number of blocks, the number of possible randomization makes brute-force approaches infeasible. Second, this approach can be applied to any ELF binary without requiring the source code of an application. Third, the randomization is performed at load time which means that potentially every execution of the binary results in a different address layout. 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.

Attack detection or prevention techniques by themselves are not sufficient as it is not clear what action must be taken once an attack is detected and also how to prevent this attack from happening again. Attack diagnosis is crucial as it not only provides input for the attack response, but also indicates which preventive measures

need to be applied, such as fine grained code randomization. Also, diagnosis information provides deeper insights into the type and complexity of an attack which reflects on the technical expertise of the attackers. This information, thus, allows organizations to strengthen their defenses by selectively applying preventive measures not just to the target process but also to other applications that respond to traffic from similar domains. Once an attack is detected and diagnosed, appropriate response actions must be deployed. Towards this goal, we propose ROPShield, a comprehensive detection, diagnosis and response framework for defense against ROP attacks.

The detection component in ROPShield employs a run-time monitoring mechanism to detect and respond to ROP attacks in real time. These attacks alter the control flow of the target process and violate certain execution constraints. Our approach detects these attacks by evaluating these constraints at various points during a program's execution. For instance, by observing the behavior of an ROP attack, we can see that the control flow of the exploit code is different from that of an ordinary program. In a normal program, the instructions of a function execute from within its own frame on the stack. However, in a ROP execution, the instructions executed due to the attack may jump to the middle of another function and execute from another function's stack frame. Our technique leverages such observations to identify an illegal execution flow. Since we check for only certain execution invariants, our approach does not require a complete control flow graph of the target process.

Once an attack is detected, ROPShield performs attack diagnosis to identify the type and cause of this attack. The type of attack is identified based on the execution constraints that are violated. Further, the process state at the time of attack along with debugging information is used to identify the precise cause of the attack. ROPShield generates a diagnosis report containing information about the attack and the process state at the time of attack. The collection of such fine grained diagnosis information is possible because of tracing based detection technique that allows us to continuously examine the process' execution state. An accurate diagnosis also al-

lows us to deploy better responses such as identifying and patching a buffer overflow vulnerability.

We propose two types of response mechanisms, both of which must be deployed to effectively secure any system. The first type of response is the *immediate response* that must be deployed as soon as the attack is detected to prevent further damage. Examples of such response include terminating the process, shutting down the system or blocking the IP address that was the source of this attack. While this type of response offers immediate protection, it does not solve the problem in the long run as the attacker might be able to replay the original or slightly tweaked version of the attack as soon as the system is live again.

The second type of response, *long-term response*, is aimed at preventing such attacks from occurring in the future. The critical component of this response is to identify the software vulnerability that led to the injection of ROP exploit payload and patch this vulnerability. In current systems, this is usually done manually by security administrators, or other specialized staff, that analyze the system logs to identify the cause of the attack and the vulnerable buffer that led to the buffer overflow. This is clearly a time consuming and error prone approach. It is desirable to automate this type of response where the vulnerability is automatically identified and patched without significant effort from the security administrators. In this respect, our tool leverages the diagnosis report generated during attack diagnosis phase to identify and patch the vulnerability. Our technique uses a combination of tools and techniques (as discussed in section 6.3) to identify the vulnerable buffer, fix the source code, and then restart the application. The advantage of ROPShield is that it provides an end-to-end defense by using fine grained diagnosis to seamlessly integrate appropriate response techniques without requiring significant effort from system administrator.

1.4 Document Structure

The rest of this document discusses the above topics in further detail. In the next chapter, we survey the state of the art and present the background information necessary to understand this work. In chapter 3, we discuss our profiling framework that automatically infers the familiarity and safety of a context and configures security policies accordingly. In chapter 4, we present a formal proximity model for RBAC systems and discuss proximity in five different realms - geographical, attribute-based, cyber, social and temporal proximity. Next, we present a fine grained randomization technique to defend against code reuse attacks in chapter 5. In chapter 6, we present a defense framework against ROP attacks that integrates the components for detection, diagnosis and response against these attacks. In chapter 7, we summarize the work done in this dissertation.

2 STATE OF THE ART

This chapter presents a survey of state of the art research work relevant to the topics discussed in this document. We categorize this related work into three categories as discussed below.

2.1 Context Profiling and Automatic Policy Configuration

Location, WiFi and Bluetooth traces provide rich context information and have been utilized for several contextual applications. The Jyotish framework [5] utilizes the joint WiFi and Bluetooth traces for predicting the movement of users. It clusters the WiFi access point information to detect locations and uses Bluetooth traces to predict the most likely future contacts. Our work uses WiFi and Bluetooth traces to estimate context familiarity and safety.

Zhou et al. [6] and Nurmi et al. [7] use the location traces along with other information to identify meaningful places like home and work for their user. These meaningful places have several applications in location based services. We also exploit similar facts to identify points of interest and build up a context familiarity profile for these places.

The Familiar Stranger project [8] studies the properties and phenomenon of Familiar Stranger relationships. A *familiar stranger* is a stranger that the user repeatedly encounters but never interacts with. It uses a notion of device familiarity that is derived from the number of encounters with the stranger's device. The degree of familiarity is used to visualize the number of familiar strangers present at a specific place to the user. Unlike this work, we tie the notion of device familiarity to a given place and use it to estimate the familiarity and safety of a context.

Greenstadt and Beal [9] propose that mobile devices can utilize cues from user behavior to identify the users and make security decisions on their behalf. Jakobsson et al. [10] emphasize on the need for authentication techniques on mobile device with no or very limited user involvement. They utilize cues from user behavior like phone activity, mobility etc. to implicitly authenticate the user to the device and to provide addition assurance in sensitive transactions. Our primary focus is not on the method for user authentication, but on how to select one out of many authentication methods (with varying usability and strength) based on the safety of current context.

In [11], Danezis discusses how various social contexts can be automatically inferred for users from the social graphs around them. Privacy settings for these social contexts can be extracted based on the policy that content generated in a social context should be accessible only in that context. We focus on using device's context to configure access policies.

Conti et al. [12] propose a framework, *CRePe*, for enforcing context-related policies for smartphones that requires manual configuration of policies. Our system profiles the user's context to estimate its familiarity and automatically infer policies. Our system can be integrated with the CRePE framework to allow a user to specify policies based on context familiarity as a logical sensor in addition to other sensor values.

Kelley et al. [13] introduce the notion of *user-controllable policy learning* where the user and system refine a common policy model in an incremental manner. Their system benefits from user feedback to gradually learn and identify policy improvements. Our model also incorporates user feedback to improve the decision making process.

Edwards et al. [14] highlight the pitfalls of automating access control where the control over security decisions is removed from the user's hands and given to the system. In our approach, we do not take away the control from a user. Instead, we assist the user by suggesting policy decisions and also incorporating user feedback.

2.2 Contextual Access Control Models

Role based access control (RBAC) [15] is a permission model that grants access based on roles that users have as a part of an organization. Several extensions to RBAC have been proposed that attempt to incorporate various contextual factors while making access decisions. In this section we will present an overview of these contextual access control models.

GEO-RBAC [16] and LRBAC [17] are contextual access control models that incorporate location of the user requesting access as a factor in deciding access control. Gal et al. [18] consider temporal attributes such as time of access as a factor in decision making. STARBAC [19], Lot RBAC [20] and Atluri et al. [21] incorporate both location of the user and time of access into the access control model. While these consider some specific contextual factors, [22–24] take a more general approach by designing access control framework that can incorporate a variety of contextual factors. Our work incorporates the proximity to other users in various realms as a factor in access control decisions. SRBAC [25] and Kirkpatrick et al. [26] consider spatial and temporal constraints for mobile RBAC systems while our approach is applicable to a more general domain.

Prox-RBAC [2] extended the notion of spatially aware RBAC to consider the relative locations of other users within an indoor space model [27,28], and is the closest paper to the current work. However, Prox-RBAC relied on an intuitive, informal notion of proximity that allowed for surprising and contradictory interpretations of proximity; furthermore, Prox-RBAC focused exclusively on the geographic realm, whereas our own work is applicable to a wider range of contextual factors.

While Prox-RBAC is unique in combining proximity constraints with RBAC, it is not the first work to consider contextual similarity between users when requests are evaluated. TMAC [29] incorporates contextual information into team-based access control by actively monitoring ongoing interactions. PBAC [30, 31] models focus on efficiently granting authorizing emergency service providers in time-critical settings.

However, all of these works restrict proximity definition to only the geographic realm, unlike our own.

2.3 Defenses against Code Reuse Attacks

We now discuss various defense techniques that have been proposed to counter code reuse attacks such as ROP attacks. Some defense techniques focus on detecting and/or preventing stack overflows. By preventing a successful buffer overflow, these defenses prevent the code reuse attacks to progress. Examples of such techniques include StackGuard [32], StackShield [33] and SmashGuard [34]. LibSafe [35] prevents exploitation of vulnerable functions for buffer overflow by intercepting calls to these functions and redirecting them to their substitute versions. This has limited applicability since it does not prevent attacks that leverage vulnerable functions which are not protected using LibSafe.

Address obfuscation [36] and address-space layout randomization (ASLR) (*e.g.,* PaX [37]) are two well-known techniques for defending against code-reuse attacks. Address obfuscation and ASLR on 32-bit architectures have the same short-comings of Instruction Set Randomization (ISR) in that the small amount of randomization leaves application vulnerable to attacks [4,38]. That is, Shacham et al. demonstrated that existing randomization techniques can be defeated by brute-force. Also, information leakage can allow an attacker to learn the randomized base address of *libc* [39]. Consequently, simply randomizing the base address does not effectively block the attack. [36] suggests randomizing function blocks as one of the address obfuscation techniques; however this particular technique was neither implemented nor discussed in detail.

In chapter 5, we propose Marlin defense technique that uses fine-grained randomization to break the ROP attack assumption of predictable address layout. Some recent research works have also explored the idea of software diversification as a defense against ROP attacks. ILR [40] randomizes location of every instruction in the

application code and and guides the execution using a fall through map. ILR relies on a process-level virtual machine that incurs a performance cost throughout the duration of the application. In contrast, Marlin's performance impact is primarily limited to the start-up cost. Pappas et al. [41] propose an in-place code randomization technique that probabilistically breaks 80% of the instruction sequences that are useful for attacks. However, Marlin provides stronger guarantees by shuffling the entire memory image, thus probabilistically breaking all instruction sequences. Also, Marlin randomizes the executable with *every run* unlike [40] and [41] that do not re-diversify the binary. XIFER [42] and STIR [43] apply software diversification to an application at runtime to protect against code-reuse attacks. While [41–43] apply diversification at the granularity of basic blocks, we randomize at function block level and show that this is sufficient to make brute force attacks infeasible. Also, these techniques would incur more overhead than Marlin as they randomize at a very fine granularity. Marlin is a novel solution for thwarting ROP attacks and does not have the limitations discussed above.

Another work that uses similar methodology as Marlin is ASLP [44]. However, this work substantially differs from our work in intent, requirements and low-level techniques. ASLP requires user input, while Marlin works without user input. ASLP requires relocation information, without which the program has to be recompiled and relinked. It involves rewriting ELF header, program header and section headers and shuffling around sections in addition to functions and variables. We randomize only the function blocks within the code segment and show that it introduces sufficient entropy to thwart ROP attacks. Thus, our approach incurs less overhead than ASLP. Bhatkar et al. [45] also propose a randomization approach to protect against memory error exploits. However, the technique used by them differs from Marlin since they associate a function pointer with every function and transform every function call into an indirect function using this function pointer while we perform binary rewriting. Also, unlike Marlin, function reordering in [45] is not done at load time.

Marlin can be seen as a variation on the idea of proactive obfuscation [46]. 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. However, Marlin has more general applicability than to replicas in distributed services. Some techniques such as [47, 48] reorder functions for performance optimization at linking stage. Since the output of these approaches is just one optimized binary, they do not diversify the binary and hence do not offer any strong protection against ROP attacks.

DynIMA [49] 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 [50, 51]. ROPecker [52] detects ROP attack at runtime by checking the presence of long chain of gadgets in the past and future execution flow. The disadvantage of these approaches is that there is a non-zero performance cost for every checked instruction. Also, with the exception of [51], these schemes assume gadgets end in `ret` instructions, and do not consider the more general case where gadgets may end in jump instructions.

Compiler-based solutions [53, 54] that create code without `ret` instructions have also been proposed. G-Free [53] is a compiler based approach that eliminates free-branch instructions and prevents mid-instruction jumps. 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 [55]. 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 promis-

ing, it is not universally applicable, and would not stop an attacker with a singular target. Further, it would require access to application's source code that is not typically available.

Control flow integrity (CFI) [56] based defenses ensure that the program execution conforms to the pre-determined control flow graph (CFG) [56–58]. Instead of checking for control violation before an instruction execution, control flow locking [58] performs lazy checks for control flow violation after a control transfer has occurred. [57] performs dynamic integrity checking by using binary instrumentation to detect code reuse attacks. Total-CFI [59] detects control flow exploits by enforcing system wide control flow integrity. CFI based approaches generally require complete CFG information which is not always available. We have proposed a runtime detection technique, ROPShield, that does not have this limitation. CCFIR [60] is a recently proposed protection against control hijacking that limits indirect control transfers to pre-collected legal targets. However, return to libc attacks are still possible on CCFIR-protected binaries if the target function (example, `system()`) is a part of the legal target. Our approach ROPShield does not require CFG information as it checks for execution invariants that are independent of specific program behavior. Also, our technique is based on tracing and does not perform binary instrumentation or source transformation.

Dynamic detection techniques based on binary instrumentation have also been proposed [50, 61]. DROP [61] is a binary monitor implemented as an extension to Valgrind [62]. 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.

It is important to notice that all the above approaches only deal with detection and/or prevention of control hijacking attacks. They do not provide any diagnosis

information or response framework. Our system, ROPShield, integrates detection, diagnosis and response components into a single framework, thus providing a complete defense approach. Some other existing approaches also provide diagnosis information. PointerScope [63] captures the key attack steps by identifying pointer misuse. Misuse is detected as type conflicts when other types of data are used as control pointers. However, this work does not integrate any response mechanism to show how the attack steps that were identified can be used in deploying appropriate response. DIRA [64], implemented as a GCC compiler extension, transforms the target program's source code so that it can detect a control hijacking attack, repair the memory damage and identify the attack packets. The only response mechanism deployed by DIRA is memory repair, while ROPShield framework integrates multiple response mechanisms. Other patching techniques [65–67] have been proposed as well that automatically identify and/or patch vulnerable buffers responsible for buffer overflow. Another work with similar goal to ours is SafeStack [68]. SafeStack uses memory access virtualization technique which relocates vulnerable buffer to a protected memory region. Once this patch is applied, the application survives future attacks on this vulnerable buffer. Unlike ROPShield, this approach does not diagnose the type of attack (such as ROP) and proposes a single response action, that is patching the vulnerability. Our approach integrates additional response mechanisms such as enabling code randomization and these response actions are configurable. ROPShield shares some similar goals with DIRA and SafeStack but adopts a different methodology.

3 INTUITIVE SECURITY POLICY CONFIGURATION

Configuring security and privacy policies in mobile devices can pose major usability challenges for the end user. Often, the difficulty in understanding the configuration options and choosing the correct settings for access control mechanisms discourages users from using those mechanisms in the first place. In this chapter, we describe the design of a context profiling framework to intuitively infer sensible access policies without user intervention, while still allowing corrective user feedback. We use the device lock scenario as an example of applying our context profiler. However, context-profiling based approach is not limited to device-locking and has several other use cases as discussed in [69]. We describe the implementation architecture for the context profiler. We then describe several experiments using a previously available dataset based on which we select concrete parameters for our prototype of the context profiler. We provide an evaluation of our model and discuss limitations and possible enhancements.

3.1 Concepts and Design

3.1.1 Context Profiling

Detecting Contexts of Interest

A *context of interest* (CoI) represents a context that is significant to the user. In this chapter, we limit our scope to geolocational contexts only. We use a grid-based clustering algorithm for GPS observations to detect CoIs, which are regions where the device has been present sufficiently often. A CoI is represented by a circular region with a fixed radius centered at the centroid of the locational observations contributing to the CoI. Once a CoI is detected, we update its centroid with every new

observation that falls within the CoI. It should be noted that the clustering module can be replaced by other sophisticated clustering schemes (such as density based clustering) that detect contexts of arbitrary shapes. Our profiling framework only requires a mapping of current position to a CoI so that it can maintain a familiarity profile of that CoI. We choose grid based clustering since it is lightweight and efficient.

Device Familiarity

A user may observe certain devices more often than others in a given CoI. These devices gradually become familiar to the user's device with respect to that particular CoI. We introduce the notion of familiarity of a device in a given CoI (hereafter *device familiarity*) as a measure of how frequently and how recently a device has been observed by the user's device in a given CoI. If a familiar device stops appearing in a CoI for a long time, its device familiarity should gradually decrease. Since we do not know if the device has left the CoI permanently or is temporarily absent, the device decay should be slow and gradual. This is achieved by growing the device familiarity of device d in CoI $\mathcal{C}$ with every observation of $\mathcal{C}$ that includes d, but decaying d only if it has not been observed in N_0 successive observations of $\mathcal{C}$, where N_0 is a suitably chosen constant. We capture this behavior of device familiarity using a variation of exponential moving average function as represented by equation 3.1 below.

Definition 3.1.1 ***Device familiarity*** *of a device d, with respect to CoI $\mathcal{C}$ after n observations of $\mathcal{C}$ is:*

$$DFam(d, \mathcal{C}, n) = \alpha_D * occ(d, \mathcal{C}, n) + (1 - \alpha_D) * DFam(d, \mathcal{C}, n - 1)$$

where,

$$occ(d, \mathcal{C}, n) = \begin{cases} 1 & \textit{if } d \textit{ is observed in } \mathcal{C} \textit{ in the } n^{th} \textit{ sample,} \\ 0 & \textit{if } d \textit{ is not observed in } \mathcal{C} \textit{ in the } n^{th} \textit{ sample,} \\ & \textit{and } (n - N_{last}) \mod N_0 = 0. \\ DFam(d, \mathcal{C}, n-1) & \textit{otherwise.} \end{cases} \tag{3.1}$$

where d was last seen in the ${N_{last}}^{th}$ sample of $\mathcal{C}$.

The selection of the smoothing factor α_D determines the weight $1 - \alpha_D$ assigned to the old device familiarity value in computing the new device familiarity value. For example, for a device present in every observation made in a CoI, higher values for α_D would imply quicker rise in the device familiarity value.

Context Familiarity

We estimate the familiarity of a CoI using two measures: *instantaneous familiarity* and *aggregate familiarity*. Instantaneous familiarity is an estimate of the familiarity of the CoI the user's device is currently in, in terms of the device familiarity values of the devices present in the CoI at that instant. Aggregate familiarity represents the "usual" or "typical" familiarity of a CoI over time.

Instantaneous familiarity is computed as a weighted average of the observed devices with their device familiarity values constituting the corresponding weights. The intuition is that the contribution of a device towards instantaneous familiarity of a CoI should be proportional to its device familiarity in that CoI. We compute the instantaneous context familiarity separately for each class of devices and combine them by taking the average over all device classes. Currently we consider two classes of devices: Bluetooth and WiFi.

Definition 3.1.2 ***Instantaneous familiarity*** *of a CoI $\mathcal{C}$ at its n^{th} observation can be defined as*

$$instFam(\mathcal{C}, n) = \frac{1}{|T|} \sum_{t \in T} instFam(\mathcal{C}, n, t) \tag{3.2}$$

where T is the set of device classes,

$$instFam(\mathcal{C}, n, t) = \frac{1}{|D_{\mathcal{C},n,t}|} \sum_{d \in D_{\mathcal{C},n,t}} DFam(d, \mathcal{C}, n)$$

and $D_{\mathcal{C},n,t}$ is the set of devices of class $t \in T$ observed in $\mathcal{C}$ at its n^{th} observation.

Aggregate familiarity of a CoI represents its "typical" familiarity and is computed as an exponential moving average of instantaneous familiarity.

Definition 3.1.3 ***Aggregate familiarity*** *of a CoI $\mathcal{C}$ after n observations of $\mathcal{C}$ is defined as:*

$$aggFam(\mathcal{C}, n) = \alpha_C * instFam(\mathcal{C}, n) + (1 - \alpha_C) * aggFam(\mathcal{C}, n-1) \tag{3.3}$$

where $0 \leq \alpha_C \leq 1$ is a suitably chosen constant.

The smoothing factor α_C determines how fast the aggregate familiarity should react to the changes in instantaneous familiarity. A higher value implies quicker reaction. In section 3.3, we discuss the choice of α_D and α_C values used in equations 3.1 and 3.3.

Notion of Null Device

An interesting question is how to interpret the instantaneous familiarity when no device is observed in a context. We model the absence of any other device by introducing the notion of a *null device* for each class of devices. A null device is introduced when no other device in that device class is observed. The device familiarity of a null device is computed in just the same way as for a real device using equation 3.1. Thus, in CoIs where absence of other devices is the norm, the null devices will have a high device familiarity which in turn leads to familiarity of the CoI to be high when no devices are present. On the other hand, in other CoIs, the familiarity of the null device will be low which causes the familiarity of the CoI to drop when no other devices are observed.

Inferring Context in absence of GPS Fix

Sometimes, especially indoors, the device may fail to get a GPS fix. But we still need to infer the current context since access control has to be enforced. WiFi- and cell-tower-based localization is typically used for positioning in the absence of GPS. This requires the device to scan the neighborhood for WiFi access points or cell-tower identifiers and map them to a geospatial location with the help of a central server. Given our design principle of not allowing any context data to leave the device, we prefer not to rely on server-assisted positioning.

Instead, we use purely local mechanisms to infer the user's context. The snapshot of (stationary) WiFi devices observed in a CoI is fairly static and can be used to attribute user's current position to a known CoI. We also leverage the fact that the instantaneous WiFi familiarity score of a CoI represents how familiar the current snapshot of WiFi devices is to this CoI to map a WiFi snapshot to its most familiar CoI.

Inference of user's context is done in two steps. First, we compute candidate instantaneous WiFi familiarity for the current snapshot of WiFi devices with respect to all known CoIs for the user. We use a minimum threshold for WiFi instantaneous familiarity to discard obviously incorrect CoI choices. The current position is then attributed to the CoI with maximum WiFi instantaneous familiarity score. If none of the candidate instantaneous familiarity scores exceed the minimum familiarity threshold, we use Jaccard's distance measure to compute the distance between the current snapshot of WiFi devices and the snapshot of WiFi devices corresponding to the last known observation with an associated GPS reading. If the two WiFi snapshots are close enough, we attribute the current observation to the same location.

From Familiarity to Safety

Familiarity can have different interpretations in terms of safety for different applications. A familiar place may be considered safe by a certain application, and unsafe

by some other application. For example, applications where anonymity is desired would treat a familiar place as unsafe and an unfamiliar place as safe. On the other hand, a configurable device lock mechanism would treat an unfamiliar place as unsafe. Perception of safety can also vary from user to user: two different users co-located in the same context may perceive different safety levels for the exact same context. In other words, how best to infer the safety level from the familiarity estimates is a difficult question. Below, we outline the current, somewhat simplistic, approach we have taken for mapping from familiarity to safety. This remains an active area of current work for us.

We propose a familiarity to safety mapping for device lock and other applications with similar requirements. For device lock, we need to assess the safety level of the current context of the device so that the appropriate locking timeout and unlocking method can be enforced. We define the security model for device lock application as follows. The goal is to prevent anyone other than the owner from misusing the device in an unlocked state. This can be done either by a thief who has stolen a device or a curious individual. Misuse of device may involve access to personal information, installation of malware/spyware and using user's credentials to carry out transactions maliciously.

Studies [70, 71] in various contexts have shown that familiarity breeds trust and reduces the risk perception. Further, statistics reported by Bureau of Justice [72] for year 2006 indicate that at least 59.2% of theft crimes were performed by strangers. Thus, it seems reasonable to assume that in the case of applications like device lock, the presence of strangers implies a potentially unsafe situation. We begin with the following intuition: a CoI that has a high familiarity both typically and currently is probably safe; as a dual, a CoI that has a low familiarity both typically and currently is probably unsafe.

We incorporate the above observations in our algorithm to estimate the safety level of the current context (Figure 3.1). The algorithm uses the instantaneous and aggregate familiarity of the current CoI to estimate the safety level as one of high

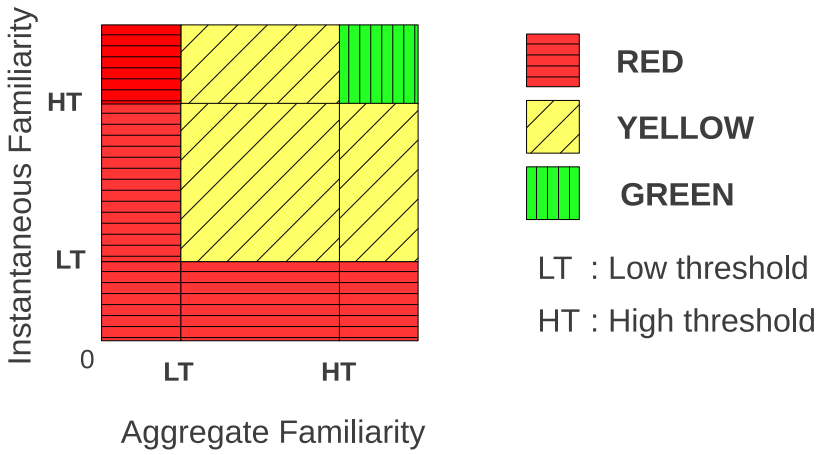


Figure 3.1. Familiarity-to-safety mappings

(GREEN), medium (YELLOW) or low (RED). To do this, we use two thresholds: a high familiarity threshold (HT) and a low familiarity threshold (LT) to delimit "high" and "low" values for familiarity (both instantaneous and aggregate). In section 3.3, we estimate reasonable values for these thresholds.

If the current context does not correspond to a CoI, we conclude that the safety level is low (RED). This is consistent with algorithm in Figure 3.1 because the aggregate familiarity of an unknown context is zero.

Device Lock Use-case

The inferred safety level can be used to automatically configure the unlock policy for a device lock. We map each safety level to a different unlocking method and locking timeout. For instance, GREEN safety may correspond to "slide-to-unlock" method which is less secure and more usable while RED safety may correspond to a more secure PIN-based unlock method. We couple this with a *low watermark* approach to decide the unlocking method: if a device is locked in a safe context, a change in context can lock it deeper (i.e., requiring a stronger unlocking method), but the converse is not true. The unlocking method will correspond to the safety of the least safe context encountered since the device was locked. This low watermark

approach is also intended as a defense against adversarial learning: for example, if a thief steals the device from an unsafe location but leaves it in house for a day, the context profiler will eventually learn that the thief's house is a "safe" place, but that does not help the thief because he has to first unlock the device using the stronger unlocking method.

3.1.2 Handling User Feedback

In automated access control enforcement, it is important to incorporate feedback from the user in the decision making process. Since our context profiler's safety algorithm ultimately bases its computations only on a few classes of sensor inputs, it may sometimes estimate the safety level incorrectly. User feedback is important in such cases so that the inferencing process can be tweaked to match user's expectations. Similarly, user feedback can be used to shortcut the learning process so that contexts that the user knows will become eventually familiar (like her home) can be deemed familiar more quickly.

A user can provide feedback by specifying the safety level of a context as perceived by him. The user may provide feedback on the long-term behavior of a CoI by marking it as *'Usually safe'* or *'Usually unsafe'*. Alternatively, he may want to indicate a short-term or temporary feedback like *'Now safe'* or *'Now unsafe'* for the current CoI. When a user provides 'Usually safe' feedback for the current CoI, he is also prompted to provide 'Now Safe' feedback, if appropriate. This provides a quick boost to the short term safety value.

Feedback classification: The safety feedback provided by a user can be broadly classified into following categories:

1. *Learning phase feedback:* This feedback is provided by the user during the learning phase to shortcut learning, or to override the context profiler's estimations of the safety of a context until that context has been learned. We believe that

this would be the most frequent case where user will provide feedback. A user may provide either short term or long term feedback during learning phase.

2. *Affirmative feedback:* This refers to the scenario where the user feedback matches the context profiler's perception of safety, that is the user just re-affirms the context profiler's perception. For example, when user says that certain context is *'Usually safe'* and the context profiler has already inferred the context as safe. We can safely ignore this feedback in the computation of safety scores.

3. *Corrective feedback:* This refers to the scenario when the context profiler fails to match user's perception of safety even after it has learned the context. Corrective feedback can be either short term or long term.

We base our feedback handling approach on the following two principles:

1. The effect of feedback should be immediately visible to the user. However, it should not permanently relax the safety computations, but allow for the system to react in case of sudden drops in familiarity scores.

2. When a user provides feedback, it is regarding the safety of a context and not its familiarity. Thus, the feedback handling mechanism should only tweak the familiarity to safety mapping and not the familiarity scores themselves.

We extend the basic familiarity-to-safety algorithm presented in Figure 3.1 to incorporate user feedback. To address the above principles, the instantaneous and aggregate familiarity scores are artificially boosted according to the feedback provided. These modified scores (referred to as $instFam_F$ and $aggFam_F$ in the discussion that follows) replace the original familiarity scores used in Figure 3.1.

Long term feedback reflects on the 'typical' behavior of a context. Our intuition is that such feedback would be provided in the learning phase to shortcut the learning process. The effect of long-term feedback should correct the safety computations until the context has been properly learned. This can be achieved by combining long term

feedback and the aggregate familiarity using a dynamic weight w_{LT} that gradually fades away. We use a time decay curve to decay the value of w_{LT}.

Definition 3.1.4 *$aggFam_F$ is the feedback adjusted score that replaces the aggregate familiarity score in algorithm in Fig 3.1. It is computed as:*

$$aggFam_F(\mathcal{C}, n) = (1 - w_{LT}) * aggFam(\mathcal{C}, n) + w_{LT} * LT\_Feedback \tag{3.4}$$

where LT_Feedback indicates long term feedback, with value either 0 (*'Usually unsafe'*) or 1 (*'Usually safe'*).

The dynamic weight w_{LT} for long term feedback is computed as:

$$w_{LT} = \begin{cases} 1 - (\frac{n_f}{N_f})^c & \text{if } n_f \leq N_f \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

where n_f is the number of observations since the long term feedback was given, N_f is the maximum number of observations after which the feedback effect should wear off and c is a constant that determines the speed of decay.

The long term feedback weight should decay slowly in the beginning so that the device has enough time to learn the context and then gradually fade away to 0. The constant N_f is decided based on the length of learning period, which depends on the α_C and α_D values. One may question as to why long term feedback should be forgotten over time. Permanently overriding the profiler's decision by user's feedback prevents the profiler from reacting to genuine drops in safety of a *usually safe* CoI (for example, a party at home). Thus, we chose slow decay of long term feedback to allow adaptive measures instead of permanent override.

Short term feedback reflects on the safety of current snapshot of a CoI. It indicates a temporary change in the behavior of a CoI and should fade away after a short time. We compute this score by combining short term feedback and instantaneous familiarity using a dynamic weight w_{ST}.

Definition 3.1.5 *$instFam_F$ is the feedback adjusted score that replaces the instantaneous familiarity score in algorithm in Fig 3.1. It is computed as:*

$$instFam_F(\mathcal{C}, n) = (1 - w_{ST}) * instFam(\mathcal{C}, n) + w_{ST} * ST\_Feedback \tag{3.6}$$

where ST_Feedback indicates short term feedback, with value either 0 (*'Now unsafe'*) or 1 (*'Now safe'*).

The short term dynamic weight w_{ST} should depend on the time elapsed and the change in the snapshot of observed devices since the feedback was given.

$$w_{ST} = \begin{cases} 1 - \max\{\frac{t-t_0}{t_{max}-t_0}, \text{Dist}(S_t, S_{t_0})\} & \text{if } t \leq t_{max} \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

where t_0 is time at which short term feedback was given, t is the current time, t_{max} is time after which short term effect should wear off (we use $t_{max} = 60$ mins.), S_{t_0} is the snapshot of devices at time t_0, S_t is snapshot of devices at time t and Dist() is the distance metric, the definition of which is based on the following rationale:

- Familiar devices in S_{t_0}, but not S_t should increase the distance measure
- Unfamiliar devices in S_t but not in S_{t_0} should increase the distance measure
- Unfamiliar transient devices in S_{t_0}, but not in S_t should not increase the distance measure
- Familiar devices in S_t but not in S_{t_0} should not increase the distance measure

Let n denote the number of observations of context $\mathcal{C}$ at current time t and $\text{occ}(d, S1, S2) = 1$ if device $d \in (S1 - S2)$ and 0 otherwise. Then we define[1]

$$\text{Dist}(S_{t_0}, S_t) = \frac{1}{|S_{t_0} \cup S_t|} \times \left(\begin{array}{l} \sum_{d_i \in S_{t_0}} \text{DFam}(d_i, \mathcal{C}, n) * \text{occ}(d_i, S_{t_0}, S_t) \\ + \sum_{d_i \in S_t} (1 - \text{DFam}(d_i, \mathcal{C}, n)) * \text{occ}(d_i, S_t, S_{t_0}) \end{array} \right) \tag{3.8}$$

The effective safety level is inferred as shown in Figure 3.1 where $instFam_F$ and $aggFam_F$ will serve the purpose of instantaneous and aggregate familiarity respectively.

[1]We could define $Dist()$ simply as the Jaccard distance $J_\delta(S_{t_0}, S_t)$, but that will not distinguish devices based on familiarity.

3.2 System Architecture

The system architecture for the context profiler software is described in Figure 3.2. It consists of three main modules:

- **Data Collection** module is responsible for continuously sensing the current context and collecting raw data about various context variables
- **CoI Detection** module periodically clusters the location data collected by the *data collection module* to detect CoIs for the user, based on their significance to the user which is determined by the amount of time the user spends in a particular place.
- **Context Analysis** module is responsible for analyzing the raw data and infer familiarity and safety scores for the current context. For each CoI, it maintains a context profile to keep track of the devices that are observed in a CoI and their familiarity scores with respect to that CoI. Based on the current snapshot of the CoI, it computes instantaneous and aggregate familiarity scores using equations 3.2 and 3.3 respectively. These familiarity scores are used to infer the safety of the context as discussed earlier.

In our current implementation, the *data collection module* scans the environment every five minutes to record the GPS co-ordinates (if available) as well as the currently visible Bluetooth devices and WiFi access points. This information is stored in a database on the device itself and is used by other modules to identify and analyze CoIs. This module can be extended to sense other kinds of context variables.

For CoI detection, we used a simple grid-based clustering algorithm with a grid cell width of 250 meters. We required a cluster to have at least 1% of all observations within a time window of 30 days which corresponds to 8640 observations at our current rate of sampling. Consequently, the detection threshold of 1% ($\approx$ 86 observations) would correspond to roughly an equivalent of seven hours of observations of a place in the GPS trace data for the place to become identified by our clustering algorithm

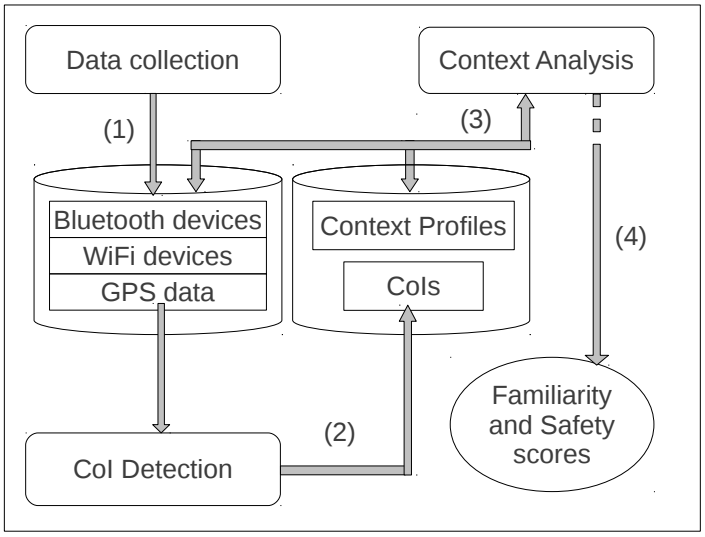


Figure 3.2. System components: (1) Data collection module collects GPS, Bluetooth, WiFi data; (2) GPS data is clustered to detect CoIs; (3) Context analysis module updates context-specific information and (4) computes familiarity and safety scores for the current context.

as a CoI. We associated Bluetooth and WiFi observations having a GPS fix within 100 meters from a cluster's centroid as belonging to that CoI. Note that a CoI is a circle with a fixed (100m) radius and is significantly smaller than a grid cell. The grid cells are used only to speed up clustering and do not dictate the size of a CoI.

The *context analysis module* periodically generates the familiarity and safety scores for the current context. These values can be used by applications to automatically configure access policies that depend on the current context. In the device locking use case, the safety scores are used to dynamically configure the unlocking method and the locking timeout of the device.

3.3 Parameter Tuning

We ran several experiments using traces from the Lausanne Data Collection Campaign, a large-scale data collection experiment focusing on mobile device users' behavioral and contextual data traces [73,74] in order to gain the insights and heuristics

needed to determine suitable parameters for a concrete instantiation of the context profiler framework. The dataset contains GPS location traces and regular scans of WiFi and Bluetooth radio environments of a large number of users.

To match our device implementation as closely as possible, we filtered the dataset to include one Bluetooth and WiFi scan observation per five-minute observation window, if available. Each of these Bluetooth/WiFi observations was matched with the closest GPS fix within the time window, if available. By applying our CoI identification algorithms, we identified a total of 167 CoIs for 37 users, giving on average 5.22 CoIs per user (median 5 CoIs).

In the device lock scenario, the context profiler effects visible to the end user are (a) how long does it take for a safe CoI to be recognized as such by the context profiler and (b) how volatile is the safety labeling of a safe CoI. As a guiding principle, we want the context profiler to learn safe CoIs within two days. At our current sampling frequency of every five minutes, a day consists of 288 observations. We conjectured that a user is likely to spend about a third of a day in a given safe CoI. Thus we need safe CoIs to be deemed safe in about 200 observations. We set this as our approximate target. We then determined suitable values for various parameters as discussed below.

Smoothing factor for Device familiarity α_D: From Equation 3.1 we see that higher values for α_D will imply that the device familiarity $DFam$ will grow quickly if a device continues to appear in successive samples in a CoI. Given our rough target of recognizing a safe CoI within 200 observations, we decided to select α_d so that a device that appears in about 20 consecutive samples of a CoI would have a $DFam$ reaching 0.9. Using Equation 3.1, we compute this value of α_D to be 0.1. This is in line with the standard practice of choosing a smoothing factor between 0.05 and 0.3 for processes that are locally constant (Chapter 8 of [75]).

Decay interval for Device familiarity N_0: To select the value of N_0 in equation 3.1, we reasoned that the familiarity of a device should decay if it did not show up even once in consecutive samples spanning a day. Again, based on the assumption

that a user may spend about a third of a day ($\approx$ 96 observations) in a given safe CoI, we chose N_0 to be 100.

Smoothing factor for Context familiarity α_C: In Equation 3.3 the smoothing factor α_C affects the lag time of the smoothing applied to the aggregate familiarity scores. The lag time determines the number of observations required for the aggregate familiarity score to react to changes in the trend of the instantaneous familiarity scores. Consequently, the choice of α_C will impact both the user-visible effects discussed above.

We presume that most users have at least two frequently visited CoIs (e.g. their home and workplace). We further assume that the majority of such CoIs can be presumed to be 'familiar' places for the users. We denote the set of the top-two most frequently observed CoIs of each user as the set of *frequent CoIs*. We studied how different choices of α_C affects the evolution of the aggregate familiarity score in frequent CoIs over time. Figure 3.3 shows the result: the y-axis on the left shows the average aggregate familiarity for frequent CoIs; the y-axis on the right shows the average of the standard deviation of the aggregate familiarity of the same, calculated over the latest 100 observations at each point. We observed the following from Figure 3.3:

- values of α_C greater than 0.05 have little impact in the behavior of the average aggregate familiarity score.
- the "knee" in the graph near the 200th observation implies that most of the frequent CoIs reach a steady state after this point.
- the average standard deviation of aggregate familiarity scores is reasonably small (less than 5%) for all values of α_C less than 0.05 beyond the steady state.

Based on these results, we chose α_C as 0.05.

Long term feedback duration N_f: The number of observations for frequent CoIs to reach steady state (200) is a suitable value for N_f in equation 3.5.

Safety thresholds HT **and** LT: In Figure 3.1, a natural value for HT is the point reached by the average aggregate score of frequent CoIs at the steady state.

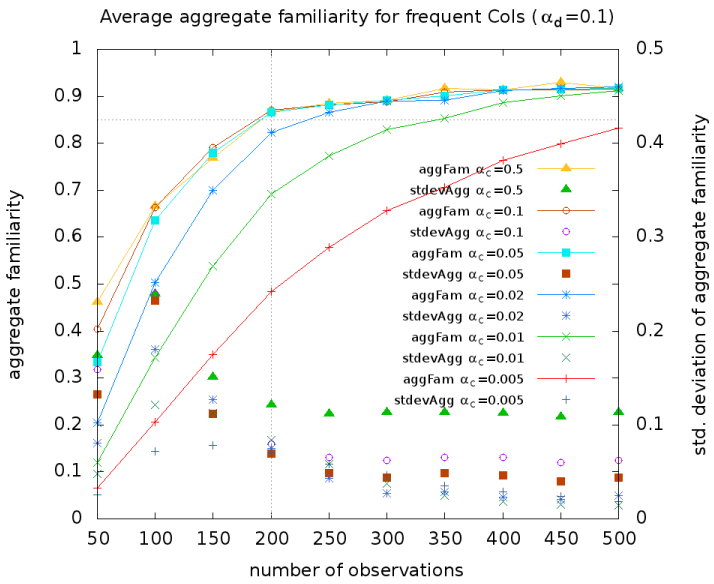


Figure 3.3. Behavior of aggregate familiarity score in frequent CoIs

From Figure 3.3, this is 0.85. To choose the value of the low threshold LT, we used the following rationale. We expect that for most users, a familiar CoI like home will exhibit stable behavior in the long-term. Thus we can choose LT such that the aggregate familiarity score of most familiar CoIs will be above this value. We resort to a 90-10 rule of thumb to assume that 90 percent of the set of frequent CoIs are likely to be stable. Figure 3.4 shows the aggregate familiarity score of the CoI at the lowest tenth percentile for a given number of observations. From the graph, we can see that 0.4 appears to be a good choice for LT because at all times after reaching the steady state (refer to Figure 3.3), all frequent CoIs in the set above the 10^{th} percentile have aggregate familiarity scores above this value.

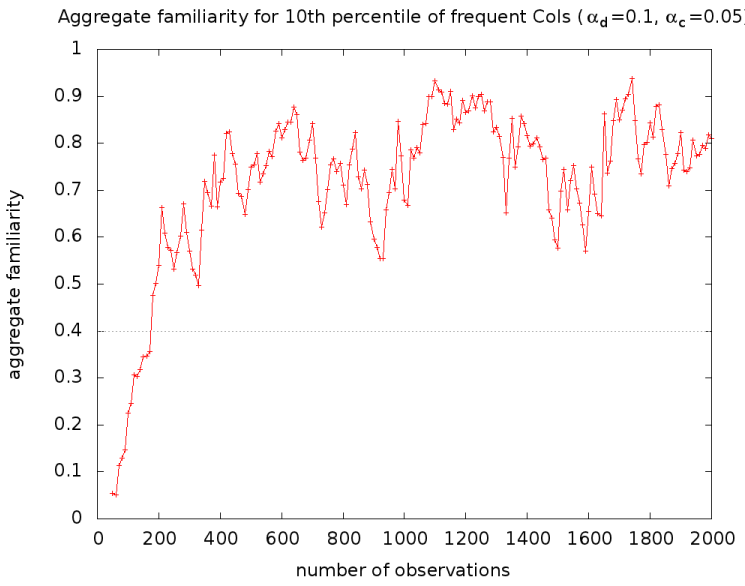


Figure 3.4. Determining the low threshold

Table 3.1.
Classifications of place labels in ground truth data

Safe	*Unsafe*
My home	Holiday resort or vacation spot
My freetime home	Shop or shopping center
My main workplace	Location related to transportation (e.g. bus stop)
	Place for indoor sports (e.g. gym)
	Place for outdoor sports (e.g. walking)
Unclassified	
Home of a friend	My main school or college place
My other work place	Other
I don't know	

3.4 Validation of the Model

3.4.1 Comparison to Ground Truth

Once the parameters were determined, we applied our familiarity and safety algorithms to the observation data related to the frequent CoIs of each user. Ideally,

Table 3.2.
Sets used in validation

Sets in ground truth data		#
Observations in "Safe" CoIs	G_{safe}	51446
Observations in "Unsafe" CoIs	G_{unsafe}	2607
Observations in Unclassified CoIs	G_{UC}	10119
Sets identified by Context Profiler		#
"Safe" observations	C_{GREEN}	55234
"Unsafe"observations	C_{RED}	2862
Neither	C_{YELLOW}	6076
Set intersections		#
True "Safe" obs.	$\lvert G_{safe} \cap C_{GREEN} \rvert$	47197
Other "Safe" obs.	$\lvert \{G_{unsafe} \cup G_{UC}\} \cap C_{GREEN} \rvert$	8037
True "Unsafe" obs.	$\lvert G_{unsafe} \cap C_{RED} \rvert$	889
Other "Unsafe" obs.	$\lvert \{G_{safe} \cup G_{UC}\} \cap C_{RED} \rvert$	1973

the evaluation of the model would be based on ground truth information indicating the user's perception of the safety of a CoI over time. Unfortunately the dataset we used did not have ground truth information at this granularity. However, it did have information where the users have labeled locations using one of several pre-defined labels such as "My home", "My main work place", "Shop" etc. We grouped these labels into "safe" and "unsafe" as shown in Table 3.1. We ignored locations with labels whose safety classification from a user's perspective is unclear (e.g., labels such as "Home of a friend"). Making the simplifying assumption that the CoIs identified by the users as "safe" or "unsafe" in the ground truth data are *always* safe or unsafe respectively, we estimated the effectiveness of the context profiler with the parameters selected above as follows. We identified the sets as in Table 3.2. Note that this labeling information we now use for the validation of the model was not part of the data we used in choosing the parameters for the model in section 3.3.

Table 3.3.
Metrics for "safe" situations

	Formula	*value*
Precision	$\frac{\lvert G_{safe} \cap C_{GREEN} \rvert}{\lvert C_{GREEN} \rvert}$	0.854
Recall	$\frac{\lvert G_{safe} \cap C_{GREEN} \rvert}{\lvert G_{safe} \rvert}$	0.917
Fallout w.r.t. "unsafe"	$\frac{\lvert G_{unsafe} \cap C_{GREEN} \rvert}{\lvert G_{unsafe} \rvert}$	0.152
Fallout w.r.t. "unclassified"	$\frac{\lvert G_{UC} \cap C_{GREEN} \rvert}{\lvert G_{UC} \rvert}$	0.755

Table 3.4.
Metrics for "unsafe" situations

	Formula	*value*
Precision	$\frac{\lvert G_{unsafe} \cap C_{RED} \rvert}{\lvert C_{RED} \rvert}$	0.311
Recall	$\frac{\lvert G_{unsafe} \cap C_{RED} \rvert}{\lvert G_{unsafe} \rvert}$	0.341
Fallout w.r.t "safe"	$\frac{\lvert G_{safe} \cap C_{RED} \rvert}{\lvert G_{safe} \rvert}$	0.019
Fallout w.r.t "unclassified"	$\frac{\lvert G_{UC} \cap C_{RED} \rvert}{\lvert G_{UC} \rvert}$	0.096

We then calculated the figures of merit for recognizing "safe" and "unsafe" situations as shown in Table 3.3 and Table 3.4 respectively.

The precision and recall of recognizing safe situations are sufficiently high. The fallout value reflecting the likelihood of unsafe CoIs receiving 'safe' classifications is slightly higher than desirable (15%), but still in acceptable range. The fallout with regard to 'unclassified' CoIs is remarkably high (75%). This may be caused by the fact that a major fraction of the CoIs in the 'unclassified' set G_{UC} actually represent places that are familiar to the user (e.g. 'Home of a friend',or, 'My other work place' might be such places). The precision of recognizing unsafe situations is low, but acceptable as it errs on the safe side. The recall is low, implying that the context profiler recognized only a third of the unsafe observations as such. However, among the 6076 YELLOW observations made by the context profiler (the set C_{YELLOW}), 1321 were in locations labeled as "unsafe" in the ground truth data. If we combine

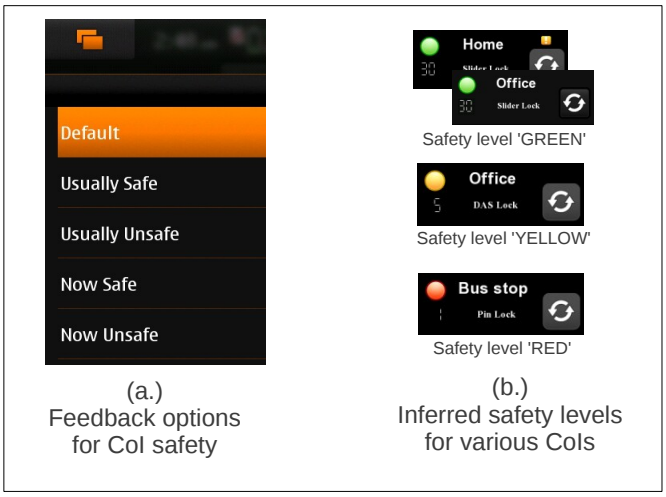


Figure 3.5. Device implementation: Feedback options and inferred safety

this set with C_{RED}, then the recall figure climbs up to 0.848. This suggests that the YELLOW safety level should not be considered significantly safer than RED. Overall, the figures of merit validate the choice of parameters.

3.4.2 Implementation

We have prototyped the context profiler with the chosen parameters on Linux-based smartphones (Nokia N900 and N9). We also implemented three different unlocking methods (passcode, draw-a-secret, and slider) which were linked to the RED, YELLOW, and GREEN safety levels respectively. The three safety levels also corresponded to three different default timeout values of 1 minute, 5 minutes and 30 minutes respectively.

3.4.3 Effect of User Feedback

We studied the effect of user feedback using our prototype context profiler. The user can provide feedback about a CoI's safety at any time to modify its behavior using a GUI as shown in Figure 3.5(a.). Figure 3.6 shows the effect of 'Usually safe'

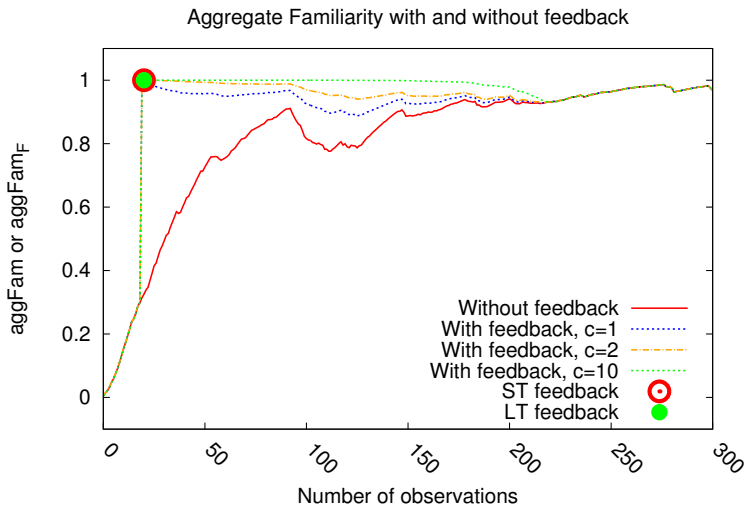


Figure 3.6. Effect of user feedback during learning

feedback that was provided by the user for a context when the context profiler was still in learning phase. The user provided 'Usually safe' feedback after approximately 20 observations of this CoI. At this point, the aggregate familiarity was artificially boosted to a value 1 ($\tilde{\text{h}}$igh) and this boost was decayed slowly. As can be seen in this figure, the effective familiarity stayed high until the CoI was learnt. Thus, the user could shortcut the learning phase by providing a long term feedback.

The graph shows the effect of using different c values in equation 3.5. While a bigger value of c provides a steady behavior until CoI has been learnt, it also reduces the CoI's tolerance to genuine drops in instantaneous familiarity. To address this tradeoff and from the behavior of $aggFam_F$ in Figure 3.6, we decide to use $c = 2$.

3.5 Discussion

Alternate safety algorithm

The safety algorithm discussed in Fig. 3.1 can be further strengthened by incorporating volatility of the CoI as a factor. CoIs that are stable (less volatile) should be less tolerant to changes in instantaneous familiarity. Even small changes should

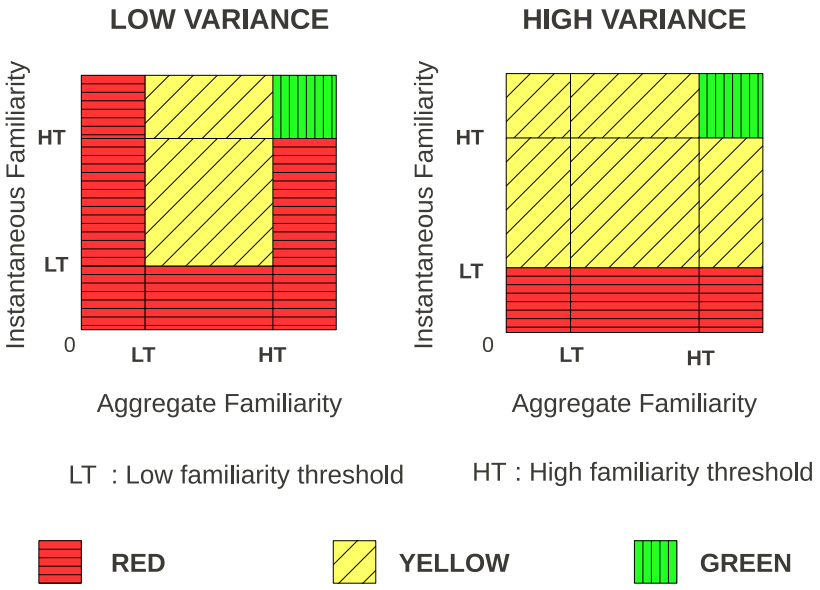


Figure 3.7. Safety algorithm with variance

severely affect the perceived safety of a stable CoI. Similarly, CoIs with high variance should be more tolerant to fluctuations. The variance of instantaneous familiarity can be an indicator for the volatility of a CoI. A context can be deemed volatile if the variance is above a certain threshold. To incorporate volatility of a CoI in the safety algorithm, we use its modified version as shown in Fig 3.7. This algorithm is not used when the user feedback is in effect, since the volatility of the CoI cannot be reliably determined in such cases.

Privacy Considerations

Collection of user's contextual data by different services usually raises *privacy* concerns. However, in our approach this data collection is used to help users in intuitive enforcement of access control and never leaves the device's storage.

Security Considerations

The *security* requirements of context profiling depends on the application. An attacker who can fake Bluetooth or WiFi addresses can influence the estimated familiarity scores. This can be addressed by revising the familiarity calculations by giving greater weights to devices whose identities are cryptographically verifiable based on existing security associations with those devices. For the device lock application this is not a significant concern because we target users like Alice (described in the Introduction chapter) who do not use any device lock in the first place. Compared to this starting point, if the use of the context profiler improves the perceived usability of device lock for such users, it can only improve the security!

Unknown Contexts

In the current implementation, as discussed earlier, an unknown context is treated as unsafe. This is logical because there is no notion of aggregate familiarity for an unknown context. However, this approach may be too pessimistic: one can plausibly make the argument that the familiarity of an unknown context where all devices present are highly familiar should be high. Since we already keep track of the number of times a device has been seen in all contexts from which we can estimate the *global familiarity* of a device and use those to estimate the familiarity of unknown contexts. The exact formulation is left as future work.

3.6 Limitations

Energy Considerations

Continuous context profiling comes at a cost of increased *battery consumption*. This limitation can be overcome by using intelligent sampling techniques. For example, instead of performing frequent GPS scanning, one could use accelerometer triggered scanning so that GPS is turned on only when motion is detected. Another

technique to conserve battery could be to use WiFi access points to detect geo-location instead of GPS. Our initial prototype does not incorporate these enhancements yet. However, intelligent sampling would be highly desirable in a usable product.

Corrective Long Term Feedback

For long term user feedback, in addition to "usually unsafe", it is reasonable to let the user assert "always unsafe" in a CoI so that that CoI is tagged as unsafe regardless of the familiarity calculation. Similarly, if we can develop a metric to measure the "similarity" of CoIs, then when a user asserts a CoI as "unsafe", that may be a cue to infer that the user may assert the same in "similar" CoIs.

Suitable Ground Truth Data

Although the analysis we performed in Section 3.4.1 gives us some confidence that our approach is valid, it suffers from the fact that the ground truth data we had available to us was not fine-grained enough. To get more accurate ground truth data we would need to conduct a targeted user study.

Intelligibility

A common concern in context-aware systems is "intelligibility" [76]: they should be able to explain to the users the bases and implications of the inferences they make. We have taken some steps towards intelligibility of the context profiler (like showing inferred safety level and the familiarity scores used in the inference), we need a more thorough analysis of how to make the context profiler more intelligible.

3.7 Conclusion

We described a context profiler which uses location traces to detect places of interest for a user and profiles the Bluetooth and WiFi devices in such places to estimate the familiarity of a place. We showed how familiarity can be used to infer safety and use this safety score to make access control decisions. Our context profiler incorporates user feedback to shortcut learning and temporarily modify the behavior of our system. We chose parameters of the context profiler by running experiments using a large dataset and evaluated the effectiveness of our approach using ground truth data from the dataset. We have prototyped the context profiler on smartphones.

Although this work focuses on a particular use case, we believe that our notion of using context profiling to infer security policies is a powerful tool. It can ease the cognitive burden on ordinary users in setting and managing appropriate security policies on mass-market personal devices. We believe that there are many other applications besides device lock that can benefit from this approach: for example, guiding the user towards context-appropriate consumption of content (e.g., warning users when they are about to open e-mail tagged as confidential while they are in a public place or when they are about to surf to a website labeled as not-safe-for-work when they are at work). We hope that this work raises the discussion on such applications and motivates fellow researchers to design them.

4 A FORMAL PROXIMITY MODEL FOR RBAC SYSTEMS

The rise of mobile and pervasive computing has made it possible to devise context-aware systems that customize the computing experience to the user's environment. One particular application for these systems is to facilitate the design of access control systems that aim to mitigate the threat of data loss by restricting permissions to appropriate settings. As these concerns are more relevant to enterprise settings, researchers often use RBAC as the foundation for designing such access control models and systems. For instance, several models have been proposed that consider the requesting user's location in the policy decision [16, 19, 21].

While such extensions to RBAC can provide a basis for reasoning about contextual policies, they fail to reflect many of the more interesting scenarios. Specifically, it may be more important to consider the relative locations of *other users*, rather than the requesting user's location. For instance, when preparing a financial deposit slip in a retail setting, the presence of a supervisor in the same room as the employee may be more important than just ensuring that the employee is present in the store office. To enable the creation of such policies, Prox-RBAC [2] was proposed to incorporate *proximity constraints* into a spatial RBAC model. That is, Prox-RBAC policies consisted of a spatial RBAC policy with an additional clause specifying constraints on the locations of other users; for instance, one can specify a constraint for a military deployment that no civilians be present. However, Prox-RBAC offered only an informal view of proximity, and unnecessarily restricted the domain to spatial concerns.

In this work, we present a more rigorous definition of proximity based on formal topological relations. In addition, we show that this definition can be applied to several additional domains, such as social networks, communication channels, attributes,

and time; thus, our policy model and language is more flexible and powerful than the previous work.

In addition to defining the model, we also present a number of theoretical results and practical advice for the creation of proximity-based RBAC systems. We propose three enforcement architectures in this work to accommodate different types of feature acquisition and communication approaches. We also provide templates for enforcement protocols for these architectures, formalize these protocols using PCL [77] and prove security properties of these protocols. In doing so, we also highlight the theoretical limits of correct enforcement of proximity constraints. We show that, given a single policy with proximity constraints, determining whether or not it can be satisfied is NP-hard. Furthermore, if the mapping of users to features can be done in polynomial time, then the problem is NP-complete. We also show that correct evaluation of proximity constraints is impossible if the deployment allows for asynchronous communication.

However, despite these results, practical deployments are still feasible. Specifically, the NP-hard result depends on the complexity of the proximity constraints; if only simple proximity constraints are used, evaluating the policy design becomes tractable. The impossibility result, on the other hand, applies to the simultaneous evaluation of the constraint for a request while another user changes features. That is, there is a small window of time when the policy information point (PIP) has an inconsistent view of the system. Note, though, that this is only relevant if the user who is changing features has some impact on the proximity constraint under consideration; if that user is not relevant to the current policy, this inconsistency has no effect on the policy decision. One can further mitigate this threat of inconsistency with redundancy; that is, by repeating the constraint evaluation after a small window of time. Thus, while perfect guarantees are impossible, the system may be able to enforce the policies well enough for practical concerns.

4.1 Concepts and Design

We begin this section by developing an intuitive understanding of proximity and realms. Once we have sketched these preliminary concepts, we define a formal proximity model and show how to map the realms to it. In doing so, we illustrate the flexibility of our model, which shows that one could adapt the same ideas to other realms of interest.

4.1.1 Intuition of Proximity

The notion of proximity can be informally defined as the nearness of two entities. These entities are active, that is they can execute actions on protected resources. Traditionally, this nearness of entities is understood in terms of physical distance, though other frames of reference, such as time, may be used. In order to use proximity as a foundational concept for access control, it is necessary to provide a formal definition that is flexible enough to accommodate various application scenarios. Before providing our definition, we will first describe five types of proximity so as to illustrate the intuition behind our formalism. Specifically, we will discuss the following types of proximity:

Geographical proximity indicates that two entities are located within a certain distance in the physical space.

Attribute-based proximity indicates that two entities share one or more common attributes, or are both located in regions of physical space that share attributes.

Social proximity indicates that two entities (represented by nodes in a social network graph) are less than a certain number of hops apart.

Cyber proximity indicates that two entities are co-present in the same online communication session.

Temporal proximity indicates that two entities are present for events separated by a limited amount of time.

Geographical Proximity

This type of proximity is perhaps the most conventional. The entities reside at specific locations in the physical world. The distance between the entities may be measured in traditional terms, such as Euclidean distance or Manhattan distance. Alternatively, the distance may be measured in logical units that are defined based on a partitioning of the reference space; for example, in an indoor space, the number of rooms separating the two entities may quantify the distance. Regardless of the measurement used, the notion of proximity implies that the distance is less than a certain threshold value. To illustrate access control based on geographical proximity, consider a policy that specifies that users must be present in the same room. A wireless sensor network could be used to track users' positions and verify that the constraint is satisfied. Another policy could specify that users must be within a certain number of meters of each other. This policy could be enforced using a technology such as Bluetooth, which indirectly vouches for the nearness of the users.

Attribute-based Proximity

In attribute based proximity model, each entity has a set of attributes that characterize certain properties or personal traits of this entity. These attributes can be encoded in credentials such as certificates that attest their validity. Attribute based proximity indicates the similarity of attributes of two entities. For example, a person with attribute 'Assistant Professor' is in attribute based proximity with another person with attribute 'Associate professor.' Weighting values can be associated with both the credential (*i.e.,* to specify its trustworthiness) or the trait itself (*e.g.,* to quantify the similarity between values). As another example, consider an online dating service where a user can choose to share his or her profile with similar users.

Potential mates with similar political views, religious backgrounds, or hometowns could be automatically granted access; such a system would be beneficial for helping users identify potential matches more quickly. In an alternate view, attributes can be associated with the user's environment, such as the type of location in the physical world. The distance metric for proximity, then, would be an empirical measure of the difference between values, possibly weighted to reflect the veracity of credentials presented. For instance, if two users are in restaurants, possible attributes may be the type of restaurant or the name of the chain; if the restaurants share the same parent company, they would be considered to be in close proximity, regardless of their physical distance. Other attributes could be the presence of public wi-fi, the temperature of the surrounding area, or the most popular professional sports team of the area. Our work allows for both uses of attributes, either relative to the user or the context.

Social Proximity

The emergent popularity of social networks introduces a new dimension to proximity. A social network is traditionally modeled as a graph where each user is represented by a node and the connections between users are represented by edges connecting them. In the social realm, the distance metric is based on the number of hops that separate two entities within the social graph. Social proximity of two user indicates that the distance between them is less than a certain number of hops. In this case, the distance is relatively static, as changes to the distance only occur when connections between users change. Although social connections may change often, it is intuitive that the distance between any two users would change more frequently in the physical world. Policies based on social proximity are quite common. The most popular is the restriction of shared data to friends or contacts. In some cases, these restrictions can be loosened to the next step in the network, such as friends of friends. In other

cases, data may be shared with other users within sub-networks; for instance, users may share data with others from the same school or employer.

Cyber Proximity

Two users are said to be in cyber proximity if they are simultaneously involved in an online communication session. For example, users may be on the same conference call or may be chatting with one another. The distance metric could be binary, indicating co-presence in the same session, or based on degrees of separation. In the latter case, consider three users named Alice, Bob, and Charlie. If Alice and Bob are chatting while Bob is connected to a conference call with Charlie, then the distance from Alice to Charlie would be two. Alternatively, if the binary metric is used, Alice and Charlie would not be in cyber proximity, as they are not present within the same communication session.

Temporal Proximity

While the previous notions of proximity can clearly be applied to users, temporal proximity means that two events occur within a certain relative time frame. The most natural metric would be the passage of units of time. However, in asynchronous systems, absolute time units may not be used or feasible. Instead, relative units, such as vector clocks, may be used to specify the ordering of events. In that case, the distance between two events would be the number of events that occur between them. An example of access control based on temporal proximity would be the specification of an expiration date on a contract signature. If another event, such as a signature by another party, does not occur prior to the expiration date, then the first signature is considered null and void. Another scenario where temporal proximity could be applied would be a combination of geo-social networks with missed connections.[1]

[1]Missed connections are popular features in publications such as alternative newspapers. One person sees another in a public place but the opportunity to meet never arises. Instead, the first person

When a user visits a public place, he may retrieve a token indicating his presence at that location at that time. This token could then be used to retrieve missed connections placed by others with the same token.

4.1.2 Hybrid Proximity Realms

Although we do not explicitly model the case, we posit that it would be possible to create policies for hybrid realms that combine two or more of the above mentioned realms. For instance, one could consider a realm that combines military ranks, the bases to which the officers are assigned, and their connections within particular social networks. Such a multidimensional policy model would combine elements of attribute, geographical and social proximities. While our model is sufficient to define such a hybrid realm (*i.e.,* by using appropriate topological relations), crafting appropriate distance metrics–by mapping realms to a multidimensional coordinate system–would be application specific. We find attempts at formalizing such a meta-model to be needlessly complex, and omit this case from further consideration for the present work.

4.1.3 Formal Proximity Model

Our formal definition of proximity is derived from constructing an abstract space model $\mathcal{S}$ from the *reference space models* or *realms* (*e.g.,* the physical world, social networks, communication sessions, time) identified in the previous section. Specifically, we apply the *calculus-based method* [3] that has been widely used in GIS applications. We start by showing that this approach is sufficient for modeling non-geographic reference space models.[2] We then show how it can be used for proximity-based RBAC systems.

---

places a missed connection advertisement with enough contextual information in the hopes that the other person will read the description and desire to make contact.

[2]While the original work only defines the method for two-dimensional geographic space, the definitions of the topological relations can be extended for multi-dimensional space, as well.

Proximity Model

Let $\mathcal{S}$ denote a discrete set of closed regions, called *features*, of the reference space model. For the feature $\lambda_i \in \mathcal{S}$, $\partial\lambda_i$ denotes the set of boundary points while λ° denotes the interior of the feature. Table 4.1 summarizes the formal definitions of these sets for each realm. For instance, in the geographical space, $\mathcal{S}$ would consist of regions of space that may or may not overlap; *e.g,* if λ_i is a room, then $\partial\lambda_i$ would be the points that constitute the walls.[3] The temporal realm would have events–closed time intervals–as features. Attribute-based proximity is similar, but extends the linear model to a multi-dimensional one. Features in the social realm would consist of sub-portions of the social network.

Before we elaborate on our model with additional definitions, we must address the complexity of the cyber realm. The difficulty lies in the fact that the most natural reference space model would be a hypergraph, with a hyperedge connecting all of the vertices (users) in the communication session, which cannot be directly mapped onto our abstract space model as it introduces inconsistencies in the topological relations. Our solution is to create a parallel hypergraph such that each vertex in the original is replaced by distinct vertices for each connected hyperedge. The interior would include the new vertices connected to the hyperedges of interest, and the boundary would be the other new vertices. For instance, if a user was simultaneously communicating in a Skype session and two chat sessions, then the feature λ_i containing the chat sessions would include the new vertices for the chat sessions in the interior, and the new vertex for the Skype session would be in the boundary.

Central to our model is the notion of *feature type*, which can be organized in a hierarchical manner. Table 4.1 provides examples of types for each realm. Types allow for system administrators to distinguish between, for instance, a physics exam and a

[3]Readers familiar with the calculus-based method will note that our abstract space model only focuses on area/area relationships. This is deliberate, as defining access control policies on single points or lines seems infeasible in general.

chemistry exam that occur simultaneously. Feature type can be either conceptual or unit-based. Conceptual feature types assign a semantic label to a feature while

Table 4.1.: Mapping of realms to abstract space model

Geographical

Elements of $\mathcal{S}$: Sets of points p in physical space

Sample types: Room, Building, Hospital

$\lambda_i = \{p \mid p \text{ is in a featured region }\}$

$\lambda_i^\circ = \{p \mid p \text{ is an interior point }\}$

$\partial\lambda_i = \{p \mid p \text{ is on the region's boundary }\}$

Attribute

Elements of $\mathcal{S}$: Attribute vectors $\overline{a} = \langle a_1, \ldots, a_k \rangle$ representing a collection of values for considered attributes. We also write $a_i \in_A \overline{a}$ to indicate a_i is one of $a_1, \ldots, a_k$.

Sample types: {Age, School}, {Age, Profession, Employer}, {Hometown}

$\lambda_i = \{\overline{a} \mid \forall\ a_i \in_A \overline{a},\ a_i \text{ is within a specified range for that attribute}\}$

$\lambda_i^\circ = \{\overline{a} \in \lambda_i \mid \forall\ a_i \in_A \overline{a},\ a_i \text{ is strictly within the specified range}\}$

$\partial\lambda_i = \{\overline{a} \in \lambda_i \mid \exists\ a_i \in_A \overline{a},\ a_i \text{ has a borderline (maximum or minimum) value for that attribute}\}$

Social

Elements of $\mathcal{S}$: Sets of edges $e \in E$ and vertices $v \in V$ such that $G = \langle V, E \rangle$ forms a social network

Sample types: Friends, Colleagues, Conference attendees

$\lambda_i = \{v \in V \mid v \text{ is an individual }\} \cup \{e = \langle v_1, v_2 \rangle \mid v_1 \in \lambda_i \vee v_2 \in \lambda_i\}$

$\lambda_i^\circ = \{v \in \lambda_i\} \cup \{e \in \lambda_i \mid e = \langle v_1, v_2 \rangle \wedge v_1 \in \lambda_i \wedge v_2 \in \lambda_i\}$

$\partial\lambda_i = \{e \in \lambda_i \mid e = \langle v_1, v_2 \rangle \wedge (v_1 \notin \lambda_i \vee v_2 \notin \lambda_i)\}$

Cyber

Elements of $\mathcal{S}$: Sets of hyperedges $\hat{h} \in \widehat{H}$ and vertices $\hat{v} \in \widehat{V}$ given a hypergraph $G = \langle V, H \rangle$ where $h \in H$ denotes a communication session and $v \in V$ denotes a user, where

$$\widehat{V} \triangleq \{\hat{v}_{i,j} \in \widehat{V} \mid \exists\ v_i \in V, h_j \in H \text{ s.t. } v_i \in h_j\}$$

$$\widehat{H} \triangleq \{\hat{h}_i = \{\hat{v}_{1,i}, \ldots, \hat{v}_{k,i}\} \in \widehat{H} \mid \exists\ h_i = \{v_1, \ldots, v_k\} \in H\}$$

Sample types: VOIP, Skype

$\lambda_i = \{\hat{h}_i \mid h_i \text{ represents a session}\} \cup \{\hat{v}_{l,i} \in \hat{h}_i \in \lambda_i\} \cup \{\hat{v}_{l,j} \in \hat{h}_j \notin \lambda_i \mid \exists\ \hat{h}_i \in \lambda_i \text{ s.t. } \hat{v}_{l,i} \in \hat{h}_i\}$

$\lambda_i^\circ = \{\hat{h} \in \lambda_i\} \cup \{\hat{v}_{l,i} \in \hat{h}_i \in \lambda_i\}$

Continued on next page

Table 4.1 – *Continued from previous page*

$\partial\lambda_i = \{\hat{v}_{l,j} \in \hat{h}_j \notin \lambda_i \mid \exists\ \hat{h}_i \in \lambda_i \text{ s.t. } \hat{v}_{l,i} \in \hat{h}_i\}$
Temporal
Elements of $\mathcal{S}$: Typed time intervals $[t_i, t_j]$
Sample types: Examination, Meeting, Football game
$\lambda_i = \{e \mid e \text{ is an event associated with some time interval } [t_i, t_j]\}$
$\lambda_i^\circ = \{t \mid t \geq t_i \wedge t \leq t_j\}$
$\partial\lambda_i = \{t_i, t_j\}$

unit-based feature types are defined by reference space such as meters (geographical), hops (social), or minutes (temporal). Realms can have multiple units, but all units would be considered to be types, and units can only be sub-types of other units; furthermore, units would be instantiated as distinct features. For instance, in a temporal space, a feature representing 8:00:00 – 8:00:59 would denote the first minute at 8:00. Let *types* denote the set of application-specific feature types for the realm, and let $\sqsubseteq$ denote a sub-typing partial order.

Definition 4.1.1 *$\tau : \mathcal{S} \to types$ denotes a* **typing function** *that maps a feature in abstract space $\mathcal{S}$ to* **feature type**. *If $\tau(\lambda_i) = t_i$, then t_i is the* **type** *of λ_i.*

The abstract space model can be restricted to only contain features that have certain types. This may be useful for applications that are only interested in certain types of features but not others.

Definition 4.1.2 *$S|_t$ denotes the* **restriction of features** *of $S \subseteq \mathcal{S}$ to those features with a sub-type of $t_j \in t \subseteq types$:*

$$S|_t = \{\lambda_i \in S \mid \exists\ t_j \in t \ s.t.\ \tau(\lambda_i) \sqsubseteq t_j\}$$

For instance, $S|_{\{exam,mathematics\}}$ would contain only time frames representing mathematics exams in a temporal discussion. In a geographical discussion, $S|_{\{room\}}$ could denote the rooms in a building.

We can now use the notion of types, in combination with topological relations, to define our abstract distance metric. We use a set of six topological relations defined in [3] to specify the relationships between features of the abstract space. Let $\mathcal{T}$ be this set of topological relations and is defined as

$$\mathcal{T} = \{disjoint, in, touch, equal, cover, overlap\}$$

We define a *connectivity chain* as a sequence of features where no two consecutive features satisfy the *disjoint* topological relation.

Definition 4.1.3 *The sequence* $\langle \lambda_0, \lambda_1, \ldots, \lambda_{n-1}, \lambda_n \rangle$ *denotes a* **connectivity chain** *from the feature* λ_0 *to* λ_n*, such that* $\neg\langle \lambda_{i-1}, disjoint, \lambda_i \rangle$ *for* $1 \leq i \leq n$*.*

Let $\chi(\lambda_i, \lambda_j)$ denote the set of all connectivity chains from λ_i to λ_j, and let $\lambda_k \in c$ mean that λ_k occurs in the chain $c \in \chi(\lambda_i, \lambda_j)$.

Definition 4.1.4 $\chi|_t(\lambda_i, \lambda_j)$ *denotes the* **restriction of connectivity chains** *connecting features* λ_i *and* λ_j *to include only intermediate features with a sub-type of* $t_k \in t \subseteq types$*:*

$$\chi|_t(\lambda_i, \lambda_j) = \{c \in \chi(\lambda_i, \lambda_j) \mid \forall\ \lambda_k \in c, \exists\ t_l \in t\ s.t.\ \tau(\lambda_k) \sqsubseteq t_l\}$$

Conceptual feature types provide logical measurement (where connectivity chain is a sequence of features). For instance, $\chi|_{\{room\}}(\lambda_i, \lambda_j)$ would only consist of chains of rooms that connect the two features. Alternatively, unit types provide physical measurement. For instance, $\chi|_{\{minute\}}(\lambda_i, \lambda_j)$ would contain chains whose intermediate features are the minutes that occur between the start of λ_i and the end of λ_j. Letting $\overline{c}$ denote the length of a chain (as measured in the number of intermediate features), we can define a basic distance metric as length of smallest connectivity chain connecting two features.

Definition 4.1.5 $\delta(\lambda_i, \lambda_j, t)$ *denotes the* **distance metric** *between features* λ_i *and* λ_j *where the intermediate feature types are restricted to* $t \subseteq types$ *and is defined as:*

$$\delta(\lambda_i, \lambda_j, t) = min(\overline{c})\ \forall c \in \chi|_t(\lambda_i, \lambda_j)$$

The final element of our proximity model is how to incorporate users. Specifically, we require some method of mapping users to features. Let $\mathcal{U}$ denote the set of users.

Definition 4.1.6 *$\mu : \mathcal{U} \rightarrow 2^{\mathcal{S}}$ denotes a* **feature mapping function** *that maps a user to set of features.*

The power set is required for the codomain as a result of the hierarchical typing of features. For instance, a user in the social realm may belong to a group of friends, as well as a group of colleagues. Hence, $\mu(u) = \{friends, colleagues\}$. It is important to note that applying μ to the temporal realm is somewhat unintuitive. From a formal perspective, μ maps that user to *all* events in which that user participated at any time. This is due to the nature of the temporal realm. In practice, the temporal μ would restrict the focus to events within a designated time frame.

Definition 4.1.7 *$\mu|_t$ denotes the* **restriction of the feature mapping function** *to types $t \subseteq$ types such that*

$$\mu|_t(u) = \{\lambda_i \in \mu(u) \mid \exists\ t_j \in t\ s.t.\ \tau(\lambda_i) \sqsubseteq t_j\}$$

Based on the preceding definitions, we can define a **proximity model** as

Role Proximity

Using the model $\mathcal{M}$, we can define the notion of **role proximity**. We start with the traditional RBAC concepts of roles ($\mathcal{R}$) and users ($\mathcal{U}$). When a user logs into the system, he is associated with a new session. Let SES denote the set of sessions, $SU : SES \rightarrow \mathcal{U}$ the mapping of sessions to users, $SR : SES \rightarrow 2^{\mathcal{R}}$ the mapping of sessions to *possible* roles that could be activated, and $Act : \mathcal{U} \rightarrow 2^{\mathcal{R}}$ the mapping of users to active roles. Observe that, for any $u \in \mathcal{U}$

$$Act(u) \subseteq \bigcup_{s \in SU^{-1}(u)} SR(s)$$

where $SU^{-1}(u)$ denotes the preimage of u under SU, *i.e.,* the set of sessions associated with the user. That is, every one of a user's active roles must be associated with some session. We can define two distinct types of role proximity using these definitions.

Definition 4.1.8 *A user $u \in \mathcal{U}$ is said to be in (t_1, d, t_2)-***weak role proximity** *((t_1, d, t_2)-wrp) of a role r for $t_1, t_2 \in types$ and $d \in \mathbb{R}^+$ if $\exists \hat{u} \in \mathcal{U}, \hat{u} \neq u$ such that all of these hold:*

1. $r \in Act(\hat{u})$
2. $\lambda_i \in \mu|_{\{t_1\}}(u)$
3. $\lambda_j \in \mu|_{\{t_1\}}(\hat{u})$
4. $\delta(\lambda_i, \lambda_j, t_2) \leq d$

Weak role proximity, then, considers only users' active roles. Observe that two feature types are necessary, as the unit separating the features will most likely have a different type than the features themselves. For instance, in social proximity, a manager at one company may be in $(org, 1, friend)$-wrp of the CTO of another company if there are employees of both companies that are friends. In a temporal setting, if a user signs a document at some meeting, $(meeting, 4, hour)$-wrp is satisfied if a manager signs the document at another meeting with no more than 4 hours separating the meetings.

At this point, it is necessary to point out that the temporal realm presents a unique complication for our definitions as written. Specifically, it is possible that r is no longer in $Act(\hat{u})$ at the time that the constraint needs to be evaluated for user u, although the proximity constraint should be considered satisfied. The solution, then, is to emphasize that $Act(\hat{u})$ is evaluated at the time that $\hat{u}$ performs some action[4]. For instance, in the preceding example, both the user and the manager must perform the action of signing the document. This modeling choice is, in essence, syntactic sugar that allows us to use consistent terminology.

[4]We note that this problem also arises in asynchronous deployments that work in different realms. However, the timing problem is heightened within the temporal realm.

This interpretation presents a clear engineering challenge, which is determining how much information about session mappings must be maintained over time. If all temporal proximity constraints require users to perform actions, then the system must only log events that occur. On the other hand, if the constraints are passive, *i.e.,* at least one of the user is not required to perform an explicit action, then the administrative overhead would be higher–possibly prohibitively high. Consequently, system designers would have to make appropriate choices for their specific applications.

Definition 4.1.9 *A user $u \in \mathcal{U}$ is said to be in (t_1, d, t_2)-***strong role proximity** *((t_1, d, t_2)-srp) of a role r for $t_1, t_2 \in$ types and $d \in \mathbb{R}^+$ if $\exists \hat{u} \in \mathcal{U}, \hat{u} \neq u$ such that all of these hold:*

1. $r \in \bigcup_{s \in SU^{-1}(\hat{u})} SR(s)$
2. $\lambda_i \in \mu|_{\{t_1\}}(u)$
3. $\lambda_j \in \mu|_{\{t_1\}}(\hat{u})$
4. $\delta(\lambda_i, \lambda_j, t_2) \leq d$

That is, strong role proximity considers roles that *could* be activated during some session for the user, but may not currently be. The rationale for strong role proximity is that it may be desirable to base policies on roles that are not currently active. For instance, if a military environment demands that there are no civilians present, strong role proximity can be used to meet this demand, because it does not require users to explicitly activate the civilian role.

Proximity Constraints

Using $\mathcal{M} = \{\mathcal{S}, \mathcal{T}, \mathcal{U}, \tau, \mu, \delta\}$ and the definitions above, we can now define **proximity constraints** that can be used in a policy language. Our language is similar to that defined in [2], except we remove the assumption of geographical proximity

and spatial roles. The simplified grammar for a proximity constraint clause can be written as:

$$
\begin{array}{lll}
C & ::- & C \vee C \\
 & — & C \wedge C \\
 & — & \neg C \\
 & — & S \quad Q \quad n \quad role \quad unit \quad thr \\
S & ::- & weak \quad | \quad strong \\
Q & ::- & at\ most \quad | \quad at\ least \quad | \quad \epsilon
\end{array}
$$

The semantics of such a constraint dictate that satisfaction requires separate users. That is, the semantics for the basic constraint ($weak\ n\ r\ unit\ thr$) dictate that there is a set $\widehat{U} \subseteq \mathcal{U}$ such that

1. $|\widehat{U}| = n$
2. $(t, thr, unit)$-wrp holds for some type $t \in types$
3. $\forall u \in \widehat{U}\ r \in Act(u)$
4. $\forall u \notin \widehat{U}\ r \notin Act(u)$

Semantics for the strong variant would replace the last two criteria as

3. $\forall u \in \widehat{U}\ \exists s \in SU^{-1}(u)$ such that $r \in SR(s)$
4. $\forall u \notin \widehat{U}\ \nexists s \in SU^{-1}(u)$ such that $r \in SR(s)$

Semantics for the other possible constraints are straightforward variations. Note that t is specified independently of the proximity constraint and is determined according to the remainder of the policy. Let $\mathcal{C}$ denote the set of proximity constraints in this language.[5]

[5]Observe that this syntax only supports a single realm per constraint. Intuitively, the syntax could be extended to specify the realm and the type t within the constraint. This would allow for complex policies that consider multiple dimensions (*e.g.,* a policy could simultaneously have spatial, temporal, and social constraints). As each realm would define its own distance metric δ, we believe this approach is feasible. However, we have not fully considered the implications of this approach, and leave such composition of proximity realms for future work.

Table 4.2.: Example policies for various realms

Geographical

Example: An officer is allowed to read a secret file only if no civilian is present within 500m and at least one senior officer is present in the same room.

$types = \{room, meters\}$, $\mathcal{O} = \{SecretFile\}$,

$\mathcal{A} = \{read\}$, $\mathcal{R} = \{SeniorOfficer, Officer, Civilian\}$

Proximity Constraints $C_1 = \langle strong, at\ most, 0, Civilian, meters, 500\rangle$,

$C_2 = \langle weak, at\ least, 1, SeniorOfficer, room, 0\rangle$

Proximity tuple $pt = \langle Officer, room, C_1 \wedge C_2\rangle$

Policy: $\{pt, read, SecretFile\}$

Attribute

Example: A dating site member can view my profile if they have same profession and are no more than 10 years older.

$types = 2^{\{profession, age\}}$, $\mathcal{O} = \{MyProfile\}$, $\mathcal{A} = \{view\}$, $\mathcal{R} = \{Member, Self\}$

Proximity Constraints $C_1 = \langle weak, \epsilon, 1, Self, \{profession\}, 0\rangle$,

$C_2 = \langle weak, \epsilon, 1, Self, \{age\}, 10\rangle$

Proximity tuple $pt = \langle Member, \{profession, age\}, C_1 \wedge C_2\rangle$

Policy: $\{pt, view, MyProfile\}$

Social

Example: A member of IEEE network is allowed to view my conference album only if he is a friend of a friend or closer.

$types = \{individual, network, hops\}$, $\mathcal{O} = \{ConfAlbum\}$,

$\mathcal{A} = \{view\}$, $\mathcal{R} = \{Self, IEEEMember\}$

Proximity Constraints $C = \langle strong, \epsilon, 1, Self, hops, 2\rangle$

Proximity tuple $pt = \langle IEEEMember, individual, C\rangle$

Policy: $\{pt, view, ConfAlbum\}$

Cyber

Example: A manager can edit a shared Google document only if he is in a GoogleTalk session with a senior manager.

$types = \{GoogleTalk\}$, $\mathcal{O} = \{document_1\}$,

$\mathcal{A} = \{write\}$, $\mathcal{R} = \{Manager, SeniorManager\}$

Proximity Constraints $C = \langle weak, at\ least, 1, SeniorManager, GoogleTalk, 0\rangle$

Proximity tuple $pt = \langle Manager, GoogleTalk, C\rangle$

Continued on next page

Table 4.2 – *Continued from previous page*

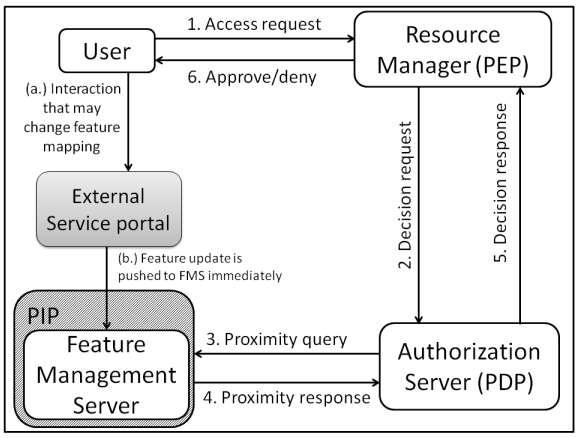
Policy: $\{pt, write, document_1\}$
Temporal Example: A supervisor can only sign an employee's time card within 24 hours after the employee did. $types = \{hours, card\ signature\}$, $\mathcal{O} = \{time\ card\}$, $\mathcal{A} = \{sign\}$, $\mathcal{R} = \{Employee, Supervisor\}$ Proximity Constraints $C = \langle weak, at\ least, 1, Employee, hours, 24\rangle$ Proximity tuple $pt = \langle Supervisor, card\ signature, C\rangle$ Policy: $\{pt, sign, time\ card\}$

Proximity-based RBAC Model

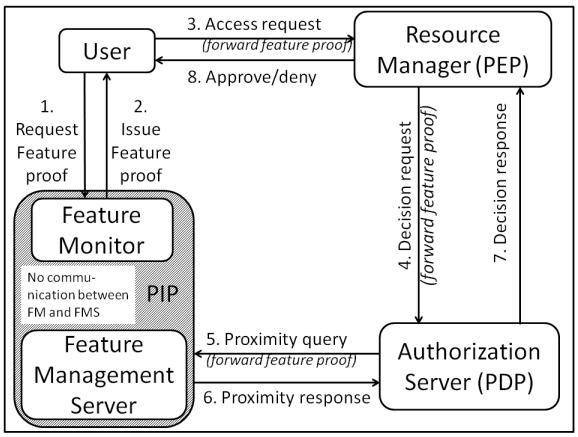
We can now conclude this section with our formal definition of a proximity-based RBAC model. Let $\mathcal{M} = \{\mathcal{S}, \mathcal{T}, \mathcal{U}, \tau, \mu, \delta\}$ denote a proximity model as defined previously. Policies would be based on **proximity tuples** $pt = \langle r, t, c\rangle$, where $c \in \mathcal{C}$ is a proximity constraint, $t \in types$ is a type associated with the requesting user's feature, and r is the requested role. Specifically, if $\mathcal{P}$ denotes the set of all such tuples, $\mathcal{A}$ denotes the set of actions, and $\mathcal{O}$ denotes the set of objects, a **proximity-based RBAC policy** would be the relation $Pol : \mathcal{P} \times \mathcal{A} \times \mathcal{O}$. That is, a policy specifies the actions allowed on an object, such that the proximity constraint (which includes the subject's role) is satisfied. The **proximity-based RBAC model** Φ would consist of the set of all such policies. Table 4.2 presents examples of policies for the five proximity realms.

4.2 Enforcement Architecture

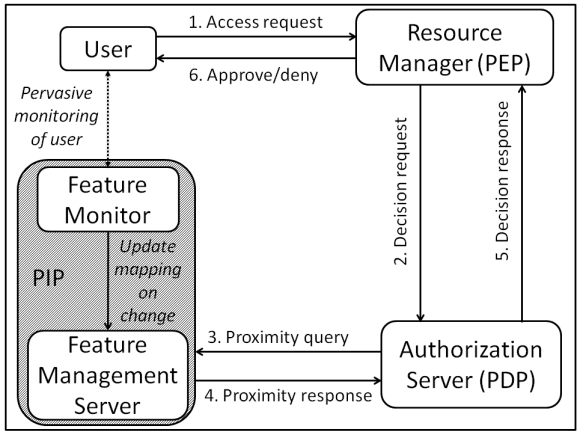
Designing a generic architecture that works across different applications and realms is crucial but challenging task. Different types of proximity and organizational



(a) No FM, External portal



(b) Independent FM, No communication



(c) FM direct communication

Figure 4.1. Enforcement architecture

settings have different requirements and a single architecture may not work for all cases. However, if an architecture is defined carefully then a major part of it can be common and only a small portion of it may need to be changed across realms. For instance, the method for acquiring feature mapping for a user is realm-specific. While different application scenarios will employ different technologies, our goal in this section is to highlight common features of principals and define required behaviors. The purpose in defining such an abstract architecture is to establish a framework for reasoning about the feasibility of designing and building proximity-based RBAC systems. We propose a generic architecture and discuss changes that are required in it to accommodate different types of feature acquisition and communication strategies.

For simplicity, we assume a centralized server with universal knowledge of the user-feature mapping. In our current approach, we emphasize the necessity of correctly mapping each user to a feature (or a set of features) in the reference space model. We refer to this process as *feature attestation*. Feature attestation could be accomplished using cryptographic techniques, such as digitally signed proofs of location, timestamps, or credentials. The central server serves as the Policy Information Point (PIP) [78] and is responsible for evaluating the proximity constraints. The Policy Decision Point (PDP) uses the result of this constraint evaluation to facilitate the proper functioning of the Policy Enforcement Point (PEP). The main components of our architecture are as follows.

- *User*: The *User* represents the entity that is assigned roles and initiates access request.

- *Feature management server (FMS)*: This server maintains the current feature mapping of every user in the system. Given a *proximity query*, in which the authorization server (see below) submits a proximity constraint and the requesting user, FMS computes the proximity distances and determines if the constraint is satisfied. It serves as the main component of the PIP and responds to queries from the authorization server.

- *Feature monitor (FM)*: This optional component is used to communicate with the user as a means of maintaining the feature mapping. If present, this component may issue a *feature proof* to the user, such as a digital certificate avowing the claimed feature, that the user can submit as a credential along with the request as shown in Figure 4.1(b). In an alternate architecture (Figure 4.1(c)), FM may pervasively monitor the user and communicate with the FMS to ensure the user-feature mapping is updated in a timely manner. Alternatively, this monitor may be absent entirely, in which case the user would communicate with an external portal that pushes feature update to FMS as discussed later (Figure 4.1(a)).

- *Authorization server (AS)*: This serves as PDP and is responsible for evaluating policy. It consults FMS by issuing proximity queries. Using the results of the queries, it evaluates the remainder of the policy and determines if the request is to be granted.

- *Resource manager (RM)*: The resource manager serves as the PEP and is responsible for controlling access to protected resources. The resource manager may hold the resources itself, or it may serve as a ticket-granting service.

In some cases, an external *service portal*, which is a trusted third party, can replace the FM. For instance, in social proximity, the proximity-based RBAC system may rely on an independent social network service for feature attestation. That is, the proximity-based RBAC system consults the external social network and overlays the feature mapping on top of the existing network. In these types of cases, the user interacts with the service portal to make changes, and the service portal pushes these updates to the proximity-based RBAC system. This is the architecture shown in Figure 4.1(a).

4.2.1 Feature Acquisition and Communication

Most of the interaction between principals is straightforward and functions like a typical RBAC architecture that consists of users, PEP, PDP, and PIP. What is unique about proximity-based RBAC is the acquisition and communication of feature mapping that is achieved via the interaction between users and the PIP. Although the precise interaction would be application specific, we identify three fundamental approaches that are illustrated in Figure 4.1 and discussed below.

No FM, External Portal

In this approach, illustrated in Figure 4.1(a), the user explicitly interacts with an external service portal (*e.g.,* a social network website or a trusted third-party attribute certification service) that is independent of the proximity-based RBAC system in order to update his or her associated feature(s). This feature update is immediately pushed by the external service portal to the FMS so that the FMS can correctly evaluate proximity constraints.

For instance, in social proximity, the user makes changes to his or her profile in a social network application, and these changes are pushed to the FMS by the application. In temporal proximity, events are logged by some application, and the FMS receives this data accordingly. This approach is applicable for all realms, though the geographical realm is challenging, as users typically do not have to interact with a centralized software portal in order to move. Instead, the other two approaches more accurately describe approaches for geographical proximity.

Independent FM, No Communication

In this approach, users interact with a distributed set of entities (feature monitors) that have no direct communication links to the FMS. These feature monitors provide the user with a credential (feature proof) that asserts the correct feature mapping.

User includes this feature proof in access request and can be validated by the FMS as shown in Figure 4.1(b).

For instance, in geographical proximity, the user may have a Bluetooth-enabled device that exchanges data with a receiver as the user moves. As the user moves, the credential updates are performed locally, and only pushed to the FMS when the user makes a request. As such, in order to enforce proximity constraints correctly, the system must force users to push their credentials sufficiently often. For instance, in the geographic realm, doors separating rooms may be considered objects. Thus, in order for the user to change features (*i.e.,* move from one room to another), he must push his credentials before the door can be unlocked.

FM Direct Communication

In this approach, a distributed sensor network (FM) continually monitors changes to the user's feature mapping. When the mapping changes, the sensor pushes the updated information to the FMS accordingly (refer Figure 4.1(c)). This approach is more appropriate for real-time geographical proximity where the location of user is pervasively tracked. This approach is also good for temporal proximity. For instance, the sensor may consist of a program that monitors updates to event log files and sends updates when the file changes.

In all above communication protocols, FMS is responsible for evaluating if the proximity constraints are satisfied. In the subsequent section, we present an algorithm for evaluating these constraints and discuss its complexity. Further, we prove some properties of this system and discuss its limitations.

4.3 Enforcement Protocols

In the generic architectures proposed in the previous section, the FMS serves as PIP and evaluates the proximity constraints. We now present an algorithm (refer Algorithm 1) that describes one approach to evaluating a single complex proximity

constraint. Specifically, assume the constraint $c \in \mathcal{C}$ consists of m primitive weak proximity constraints, each of the form (*weak n role unit thr*).[6] This algorithm will evaluate each primitive constraint to yield a Boolean value to replace the constraint. Once all constraints are evaluated, the resulting Boolean expression is evaluated. If the return value is **true**, then the constraint was satisfied. Note that handling variations allowed by the policy language involves trivial changes that do not affect the complexity of the algorithm.[7]

4.3.1 Complexity Analysis

Let $\mathcal{D}$ denote the decision problem that answers whether or not a proximity constraint can be satisfied. That is, assume μ maps a user to a feature in polynomial time. Then $\mathcal{D}$ takes as input $\mathcal{M} - \mu$ (*i.e.,* the model with no current mapping of users to features) and a policy $pol \in \Phi$. $\mathcal{D}$ returns "yes" if there exists a mapping μ such that the proximity constraint $c \in \mathcal{C}$ in the tuple $\langle r, t, c \rangle$ for the policy *pol* is true. If no such mapping exists, $\mathcal{D}$ returns "no."

Lemma 4.3.1 *Given a candidate mapping μ and a distance function δ that run in polynomial time, verifying μ satisfies pol can be done in polynomial time. That is, $\mathcal{D}$ is in NP.*

Proof Let n be the maximum of $|\mathcal{U}|$, $|\mathcal{R}|$, and m, where m is the number of primitive constraint clauses in *pol*. Executing Algorithm 1 without the mapping $\mu|_t(t)$ can be done in $\mathcal{O}(n^3)$ time. □

Theorem 4.3.1 *$\mathcal{D}$ is NP-hard.*

Proof Our proof is by reduction from Boolean satisfiability (SAT). Given an arbitrary Boolean expression, one can replace each independent variable with a unique

[6]For simplicity, we ignore the use of parentheses to shape the Boolean expression.

[7]For instance, supporting *at most* and *at least* requires adding **else** checks to the final **if-then-else** block. These cases can be enumerated and do not vary with the size of n.

Algorithm 1: Evaluate (*weak n role unit thr*) constraints

Input: $c \in \mathcal{C}$: a proximity constraint, consisting of m primitive constraints, joined using Boolean connectives ; $u \in \mathcal{U}$: the requesting user ; $t \in \mathcal{T}$: requesting user's feature type

Output: **true** or **false**

```
/* break c into its primitive constraints */
```

$\langle c_1, \ldots, c_m \rangle \leftarrow c$

$Feature_u \leftarrow \mu|_{\{t\}}(u)$

$ActiveRoles \leftarrow \emptyset$

for $c_i := c_1$ **to** c_m **do**

- $Matches \leftarrow 0$
- **foreach** $\hat{u} \in \mathcal{U} - \{u\}$ **do**
  - `/* weak proximity semantics */`
  - **foreach** $r \in Act(\hat{u})$ **do**
    - **if** $r = c_i.role$ **then**
      - $Feature_o \leftarrow \mu|_t(\hat{u})$
      - $distance \leftarrow \delta(Feature_u, Feature_o, c_i.unit)$
      - **if** $distance \leq c_i.thr$ **then**
        - $Matches \leftarrow Matches + 1$
- **if** $Matches = c_i.n$ **then**
  - $b_i \leftarrow$ **true**
- **else**
  - $b_i \leftarrow$ **false**

return EvaluateBoolean $\langle b_1, \ldots, b_m \rangle$

primitive proximity constraint in polynomial time. Based on the complexity of SAT [79], $\mathcal{D}$ is NP-hard. □

Corollary 4.3.1 *Given a candidate mapping μ and a distance function δ that run in polynomial time, $\mathcal{D}$ is NP-complete.*

Proof From Theorem 4.3.1, $\mathcal{D}$ is NP-hard. Under the assumption of polynomial run time for μ and δ, by Lemma 4.3.1, $\mathcal{D}$ is in NP. Thus, it is NP-complete. □

These results illustrate a warning for building and maintaining proximity-based RBAC systems. Clearly, the latter result shows that attempting to build an automated tool that determines if a set of policies can be evaluated would require heuristics to be tractable. Furthermore, the complexity of Algorithm 1, while polynomial-time, is not particularly efficient and may present scaling challenges. Thus, designers of proximity-based systems should plan carefully to streamline the operation of the PIP.

4.3.2 Properties of Protocols

Before describing a general approach to constructing enforcement protocols, we first present some theoretical results that illustrate the limitations of such systems. In real systems, communication between various components of the architecture may entail some delay. This communication delay may lead system into a state where the evaluation of proximity constraints at a certain time is not consistent with the current feature mapping of users. For example, the feature mapping of a user involved in a proximity constraint may change while the constraint is still being evaluated by FMS. The following results use impossibility of distributed consensus [80] to show that correct evaluation of constraints cannot be guaranteed unless FMS has correct mapping of all users and these mapping don't change until FMS has completed the evaluation of proximity constraint. Theorem 4.3.2 presents the proof for the deployment scenario illustrated in Figure 4.1(a). Corollary 4.3.1 presents this proof for the scenario in Figure 4.1(b), while the Figure 4.1(c) case is handled by Theorem 4.3.3.

Theorem 4.3.2 *Given a deployment with no feature monitor such that μ is updated only through explicit interaction with a service portal. Correct proximity constraint evaluation can be enforced only if access to the service portal (by the users and FMS) is synchronous.*

Proof Assume that evaluation can be enforced correctly. To prove that access must be synchronous, we will map proximity constraint evaluation onto a consensus protocol P. Specifically, let $p_1, \ldots, p_n$ denote asynchronous processes representing users and the service portal would consist of a buffer for P. Each p_i for a user would respond with a 1 if the user's feature has changed, 0 if unchanged, and b denotes the request is still pending. The goal of P would be to have a response of 0 for all users, indicating that the portal has the correct mapping of users to features. However, if a single p_i fails without notice (*e.g.,* the user's network connection gets dropped), then no such P can exist [80]. Thus, if users are granted asynchronous access to the service portal, no protocol involving the portal and FMS can exist that guarantees constraint evaluation is correct. Therefore, by contradiction, correct evaluation requires synchronous access. □

Corollary 4.3.2 *Given a deployment with asynchronous feature monitors, correct proximity constraint evaluation cannot be enforced.*

Proof Similar to the preceding. □

The above results address the effect of limitations of communication channel between users and FM/FMS on correct evaluation of proximity constraints. Inconsistency in constraint evaluation may also stem from asynchronicity of communication between the components of our architecture. That is, assuming that the channel between users and FM/FMS is synchronous and FMS has all correct mappings, it is still not possible to achieve correct evaluation of proximity constraints. This is because by the time the proximity evaluation decision reached RM and RM accepts/denies the request, the feature mapping of some user may have changed in a way that it changes

the outcome of proximity constraint evaluation. The following theorem proves this result.

Theorem 4.3.3 *Assuming that the communication between RM, AS and FMS is asynchronous, it is impossible for a deployment with feature monitors to guarantee correct evaluation of proximity constraints, even if the monitors have synchronous access to FMS.*[8]

Proof Similar to the preceding Lemma, except the consensus protocol is now to be executed between the principals of our architecture. That is, as communication between RM, AS, and FMS is asynchronous, these three principals cannot achieve consensus. Formally, let $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ denote a sequence of events in the evaluation of the constraint and the resulting data exchange. Assume FMS completes evaluation at σ_i and sends the result at σ_{i+1} to AS, who forwards the result to RM at σ_{i+2}. Let $\widehat{\sigma_{i+2}}$ denote the reception by FMS of a message from some FM that would change the result of the proximity evaluation. As communication between FMS and FM is independent from communication between AS and RM, $\widehat{\sigma_{i+2}}$ can occur simultaneously as σ_{i+2}. As such, when RM grants (or denies) access at σ_{i+3}, the proximity constraint may evaluate to a different value. Hence, the principals cannot achieve consensus, and correct policy enforcement cannot be guaranteed. □

We wish to emphasize that these impossibility results do not mean that one cannot build a proximity-based RBAC system that functions correctly. Rather, any such system will have brief moments when policy decisions will be incorrect. Specifically, when a user-feature mapping changes at the same time that a related constraint is evaluated, a race condition occurs. For instance, in geographical proximity, if a policy that requires the presence of a supervisor is evaluated immediately after the supervisor enters the room, it is possible that the system would have a false negative, denying access unnecessarily, as the supervisor's new location had not been propagated to the

[8]One should be careful to note that Theorems 4.3.2 and 4.3.3 are not contradictory. Rather, Theorem 4.3.2 disproves, in essence, the *converse* of 4.3.3.

FMS yet. Thus, designers of proximity-based RBAC systems should account for such cases.

4.3.3 Best-guess Protocols

Despite these impossibility results, system designers can achieve generally accurate proximity constraint evaluation, provided one can tolerate brief moments of erroneous results. We refer to this phenomenon as a *best-guess assumption*, and we provide template protocols in the next section. The general approach is that communication proceeds as illustrated in Figure 4.1. In addition, FMS stores a cache of the most recent proximity queries for continual re-evaluation over a designated period of time. The frequency of the re-evaluation would be an application-specific parameter. Within the designated time window, if the constraint evaluation changes, FMS would forward this new information to PDP. If this result changes the policy decision, PDP would inform the PEP, which would revoke access accordingly.

4.4 Heuristic-based Protocol Templates

Our aim in this section is to provide templates for enforcement protocols for architectures defined in Section 4.2. We have designed these protocols to support a number of enforcement goals, which we will formalize later. For now, our goals can be enumerated as

- validate users' claims for authorization to activate a role
- evaluate proximity constraints subject to a limited time frame
- minimize the amount of information leakage to prevent impersonation attacks
- prevent replay attacks by authorized users
- prevent improper accesses by unauthorized intruders

Our protocols employ standard cryptographic primitives. Specifically, let (Gen, Enc, Dec) denote an encryption scheme that provides indistinguishable encryptions under chosen plaintext attacks (IND-CPA-secure) such that $\text{Gen}(1^n)$ denotes a probabilistic key generation algorithm with security parameter 1^n, $\text{Enc}_k(\cdot)$ denotes encryption using the key k while $\text{Dec}_k(\cdot)$ denotes the corresponding decryption. Next, let $(Gen, Sign, Ver)$ denote a MAC scheme that is unforgeable against chosen message attacks (CMA-secure)[9].

We also adopt the standard convention that $\leftarrow$ denotes probabilistic assignment, while $:=$ denotes a deterministic assignment. Finally, while we use Enc and Dec generically, we distinguish between symmetric and public key encryption based on the key used. For instance, Enc_{K_p} refers to symmetric encryption using the key K_p for some identifier p. $\text{Enc}_{sk(p)}$ denotes encryption using the secret key of p, while $\text{Dec}_{pk(p)}$ would denote the corresponding decryption using p's public key.

In addition to standard cryptographic primitives, our protocols employ a number of additional building blocks, as follows. Recall that $\mathcal{U}$ denotes the set of users, $\mathcal{R}$ the set of roles, $\mathcal{O}$ the set of objects, $\mathcal{A}$ the set of actions, $\mathcal{S}$ the set of features in the reference space, $\mathcal{P}$ the set of proximity constraints, and Pol the set of policies. In addition, we adopt the convention that $\{0,1\}^n$ denotes a binary stream encoding some value (such as a cryptographic certificate). Lastly, as $Auth$ (authentication primitive as described below) may take more than two parameters (the first is always a user, each additional is a binary-encoded value), we use $(\{0,1\}^n)^+$ to denote the function takes one or more binary parameters.

- $Auth : \mathcal{U} \times (\{0,1\}^n)^+ \rightarrow \{true, false\}$ – a non-interactive authentication scheme that takes a user ID and one or more binary credentials as input, returning $true$ or $false$ to indicate whether the authentication succeeds

- $FindPolicies : \mathcal{R} \times \mathcal{O} \times \mathcal{A} \rightarrow 2^{Pol}$ – identifies the relevant policies for the requested role, object, action tuple

[9]For simplicity of notation, we use Gen to denote the key generation scheme for both encryption and MAC.

- $EvalTuples : (\mathbb{N} \rightarrow 2^{\mathcal{P}}) \rightarrow (\mathbb{N} \rightarrow \{true, false\})$ – evaluates a sequence of proximity tuples for the current feature mapping μ, returning a sequence of truth values declaring whether or not the associated tuple was satisfied[10]

- $Decide : 2^{Pol} \rightarrow \{true, false\}$ – determines which policies were satisfied and returns a Boolean indicating whether or not to grant access

- $Bind : \mathcal{U} \times \mathcal{R} \times \mathcal{S} \rightarrow \{0,1\}^n$ – a computationally binding procedure that produces a verifiable credential (*e.g.,* a digitally signed certificate) that ties the user to the requested role and the claimed feature at the time requested

- $GenValidation : \{0,1\}^n \rightarrow (\{0,1\}^n \rightarrow \{true, false\})$ – takes a digital credential as input and returns a function that can be applied to validate the credential at a later time

Protocol $\mathcal{Q}_0$, as shown in Figure 4.2, describes the data exchanged for Figure 4.1(a). In this architecture, the external service portal pushes updates to FMS as needed. As this portal is considered external to our architecture, communication with it is not modeled in Protocol $\mathcal{Q}_0$. Instead, Protocol $\mathcal{Q}_0$ shows the data exchanged when the request is made. We use the notation $pol_i.pt$ denotes the proximity tuple pt for the given policy pol_i. (See Section 4.1.3.) Observe that *Decide* does not declare when the decision should be made to grant access, as this is application specific. That is, some systems may require all policies to be satisfied, while others grant access if any policy is satisfied.

Protocol $\mathcal{Q}_0$ introduces several variables that may warrant additional clarification. To start, $obj \in \mathcal{O}$ denotes the object under consideration, $act \in \mathcal{A}$ is the requested action, z is a nonce, and ts denotes a timestamp. We use $cred_{feat}$ and $cred_{role}$ to denote credentials that attest to one's authorization to use a feature or activate a role.

[10]Observe that a sequence can be modeled as a partial function from the naturals to a set of items to be sorted. *E.g.,* if s is a sequence, $s(1)$ denotes the first item, $s(2)$ the second, etc.

Protocol $\mathcal{Q}_0$ – base request protocol

0) **Initialization**

$[U]\ K_r \leftarrow \mathrm{Gen}(1^n)$
$[U]\ \sigma_{kr} \leftarrow \mathrm{Enc}_{pk(RM)}(K_r)$
$[U]\ \sigma_{ur} \leftarrow \mathrm{Enc}_{K_r}(obj, act, z, ts)$
$[U]\ \sigma_{ua} \leftarrow \mathrm{Enc}_{pk(AS)}(role, id_{user}, cred_{role}, cred_{feat}, z, ts)$

1) **Access request**:

$[U \rightarrow RM]\ \sigma_{ur}, \sigma_{ua}, \sigma_{kr}$

$[RM]\ K_r := \mathrm{Dec}_{sk(RM)}(\sigma_{kr})$
$[RM]\ (obj, act, z, ts) := \mathrm{Dec}_{K_r}(\sigma_{ur})$
$[RM]\ \gamma_r := \mathrm{Sign}_{sk(RM)}(obj, act, z, ts)$
$[RM]\ \sigma_{ra} \leftarrow \mathrm{Enc}_{pk(AS)}(\sigma_{ua}, obj, act, \gamma_r)$

2) **Decision request**:

$[RM \rightarrow AS]\ \sigma_{ra}$

$[AS]\ (\sigma_{ua}, obj, act, \gamma_r) := \mathrm{Dec}_{sk(AS)}(\sigma_{ra})$
$[AS]\ (role, id_{user}, cred_{role}, cred_{feat}, z, ts) := \mathrm{Dec}_{sk(AS)}(\sigma_{ua})$
$[AS]\ valid_{req} := \mathrm{Ver}_{pk(RM)}(\{obj, act, z, ts\}, \gamma_r)$
$[AS]\ auth_{id} := \mathrm{Auth}(id_{user}, \{cred_{role}, role\})$
$[AS]\ \langle pol_1, \ldots, pol_m \rangle := \mathrm{FindPolicies}(role, obj, act)$
$[AS]\ \gamma_{af} := \mathrm{Sign}_{sk(AS)}(id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, z)$
$[AS]\ \sigma_{af} \leftarrow \mathrm{Enc}_{pk(FMS)}(id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, cred_{feat}, z, \gamma_{af}, ts)$

3) **Proximity query**:

$[AS \rightarrow FMS]\ \sigma_{af}$

$[FMS]\ (id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, cred_{feat}, z, \gamma_{af}, ts) := \mathrm{Dec}_{sk(FMS)}(\sigma_{af})$
$[FMS]\ valid_{pol} := \mathrm{Ver}_{pk(AS)}(\{id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, z\}, \gamma_{af})$
$[FMS]\ auth_{feat} := \mathrm{Auth}(id_{user}, cred_{feat})$
$[FMS]\ \langle res_1, \ldots, res_m \rangle := \mathrm{EvalTuples}(\langle pol_1.pt, \ldots, pol_m.pt \rangle)$
$[FMS]\ \gamma_f := \mathrm{Sign}_{sk(FMS)}(\langle res_1, \ldots, res_m \rangle, id_{user}, z)$

4) **Proximity response**:

$[FMS \rightarrow AS]\ \langle res_1, \ldots, res_m \rangle, \gamma_f$

$[AS]\ valid := \mathrm{Ver}_{pk(FMS)}(\{\langle res_1, \ldots, res_m \rangle, id_{user}, z\}, \gamma_f)$
$[AS]\ res := \mathrm{Decide}(\langle pol_1[res_1/pol_1.pt], \ldots, pol_m[res_m/pol_m.pt] \rangle)$
$[AS]\ \gamma_a := \mathrm{Sign}_{sk(AS)}(res, obj, act, z)$

5) **Decision response**:

$[AS \rightarrow RM]\ res, \gamma_a$

$[RM]\ valid_{res} := \mathrm{Ver}_{pk(AS)}(\{res, obj, act, z\}, \gamma_a)$
$[RM]\ retval := act[obj]$
$[RM]\ \sigma_{res} \leftarrow \mathrm{Enc}_{K_r}(retval)$

6) **Approve or deny**:

$[RM \rightarrow U]\ \sigma_{res}$

Figure 4.2. Protocol for architecture in figure 4.1(a)

To begin to analyze the security qualities of Protocol $\mathcal{Q}_0$, we can formalize the protocol using PCL [77], as shown in Figure 4.3. In PCL notation, the protocol is restructured from the perspective of various *roles*[11] that specify the behavior of various participants within the protocol. That is, instead of looking at the global view of the protocol, each participant's actions are viewed in isolation. In Protocol $\mathcal{Q}_0$, for instance, **Init** designates the initiator role. In an honest execution, the user U can take on the role of initiator, which requires knowledge of the resource manager RM in charge of the protected resource. Similarly, **Auth** is the authorization role, **Pol** is the policy management role, and **Eval** is the proximity evaluation role.

Also, note that there is a distinction between the participant of the protocol (*e.g.,* $\hat{R}$) and the associated principal (*e.g.,* RM). This distinction is important, as the participant may be an adversary attempting an attack on the system. That is, $\hat{R}$ may actually be the adversary $\mathcal{A}$. As such, the PCL specification makes this distinction obvious.

The advantage of this formalization is that it makes explicit what data is seen by each participant in the protocol, and we can infer what knowledge is gained during an execution R of the protocol. Figure 4.4 shows the knowledge gained by each participant during execution. In this notation, θ_i denotes the *a priori* knowledge of principal i, while $\phi_{i,R}$ denotes the knowledge gained from execution R. We use $\phi_{\mathcal{A},R}$ to denote the knowledge gained by a probabilistic polynomial-time (PPT) adversary with only access to the public keys of the participants. We also write $\phi_i \models \tau$ to indicate that principal i *has* or *knows* the piece of information τ. Note that $\phi_i \models \tau$ implies $\tau \in \phi_i \cup \theta_i$ or τ can be derived from some $\hat{\tau} \in \phi_i \cup \theta_i$. For simplicity, we omit any encrypted message σ_j from the $\phi_{i,R}$ sets, as our assumptions regarding encryption presume that the knowledge gain from just an encrypted message is negligible. We also omit verifications of MACs, as these are only relevant to determining the origin integrity of a message, rather than providing true information about the data exchanged.

[11] This is an unfortunate collision of terminology. The term "role" in relation to PCL should not be confused with the notion of RBAC role.

Init $\equiv (\hat{R})$ [
  new z;
  new K_r;
  $\sigma_{kr} \leftarrow$ **enc** $K_r, pk(RM)$;
  $\sigma_{ur} \leftarrow$ **enc** $(obj, act, z, ts), K_r$;
  $\sigma_{ua} \leftarrow$ **enc** $(role, id_{user}, cred_{role}, cred_{feat}, z, ts), pk(AS)$;
  send $\hat{U}, \hat{R}, (\sigma_{ur}, \sigma_{ua}, \sigma_{kr})$;
  receive $\hat{R}, \hat{U}, \sigma_{res}$;
  $res :=$ **dec** σ_{res}, K_r;
]$_{U()}$

Auth $\equiv (\hat{A})$ [
  receive $\hat{U}, \hat{R}, (\sigma_{ur}, \sigma_{ua}, \sigma_{kr})$;
  $K_r :=$ **dec** $\sigma_{kr}, sk(RM)$;
  $(obj, act, z, ts) :=$ **dec** σ_{ur}, K_r;
  $\gamma_r :=$ **sign** $(obj, act, z, ts), sk(RM)$;
  $\sigma_{ra} \leftarrow$ **enc** $(\sigma_{ua}, obj, act, \gamma_r), pk(AS)$;
  send $\hat{R}, \hat{A}, \sigma_{ra}$;
  receive $\hat{A}, \hat{R}, res, \gamma_a$;
  $valid_{res} :=$ **verify** $(\{res, obj, act, z\}, \gamma_a), pk(AS)$;
  $retval := act[obj]$;
  $\sigma_{res} \leftarrow$ **enc** $retval, K_r$;
  send $\hat{R}, \hat{U}, \sigma_{res}$;
]$_{RM()}$

Pol $\equiv (\hat{F})$ [
  receive $\hat{R}, \hat{A}, \sigma_{ra}$;
  $(\sigma_{ua}, obj, act, \gamma_r) :=$ **dec** $\sigma_{ra}, sk(AS)$;
  $(role, id_{user}, cred_{role}, cred_{feat}, z, ts) :=$ **dec** $\sigma_{ua}, sk(AS)$;
  $valid_{req} :=$ **verify** $(\{obj, act, z\}, \gamma_r), pk(RM)$;
  $auth_{id} :=$ Auth $(id_{user}, \{cred_{role}, role\})$;
  $\langle pol_1, \ldots, pol_m \rangle :=$ FindPolicies$(role, obj, act)$;
  $\gamma_{af} :=$ **sign** $(id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, z), sk(AS)$;
  $\sigma_{af} \leftarrow$ **enc** $(id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, cred_{feat}, z, \gamma_{af}, ts), pk(FMS)$;
  send $\hat{A}, \hat{F}, \sigma_{af}$;
  receive $\hat{F}, \hat{A}, (\langle res_1, \ldots, res_m \rangle, \gamma_f)$;
  $valid :=$ **verify** $(\{\langle res_1, \ldots, res_m \rangle, id_{user}, z\}, \gamma_f), pk(FMS)$;
  $res :=$ Decide$(\langle pol_1[res_1/p1.pt], \ldots, pol_m[res_m/pol_m.pt] \rangle)$;
  $\gamma_a :=$ **sign** $(res, obj, act, z), sk(AS)$;
  send $\hat{A}, \hat{R}, (res, \gamma_a)$;
]$_{AS()}$

Eval $\equiv ()$ [
  receive $\hat{A}, \hat{F}, \sigma_{af}$;
  $(id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, cred_{feat}, z, \gamma_{af}, ts) :=$ **dec**$(\sigma_{af}), sk(FMS)$;
  $valid_{pol} :=$ **verify**$(\{id_{user}, \langle pol_1.pt, \ldots, pol_m.pt \rangle, z\}, \gamma_{af}), pk(AS)$;
  $auth_{feat} :=$ Auth$(id_{user}, cred_{feat})$;
  $\langle res_1, \ldots, res_m \rangle :=$ EvalTuples$(\langle pol_1.pt, \ldots, pol_m.pt \rangle)$;
  $\gamma_f :=$ **sign**$(\langle res_1, \ldots, res_m \rangle, id_{user}, z), sk(FMS)$;
  send $\hat{F}, \hat{A}, \langle res_1, \ldots, res_m \rangle, \gamma_f$;
]$_{FMS()}$

Figure 4.3. PCL specification for protocol $\mathcal{Q}_0$

$\theta_U = \{obj, act, role, id_{user}, cred_{role}, cred_{feat}, ts\}$
$\theta_{RM} = \emptyset$
$\theta_{AS} = \{Auth, FindPolicies\}$
$\theta_{FMS} = \{Auth, (\mathcal{U} \times \mathcal{S})\}$
$\phi_{U,R} = \{retval, z, K_r\}$
$\phi_{RM,R} = \{K_r, obj, act, z, res, retval, ts\}$
$\phi_{AS,R} = \{obj, act, role, id_{user}, cred_{role}, cred_{feat}, z, ts, auth_{id}, \langle res_1, \ldots, res_m \rangle\}$
$\phi_{FMS,R} = \{id_{user}, \langle pol_1.pt, \ldots pol_m.pt \rangle, cred_{feat}, z, ts\}$
$\phi_{\mathcal{A},R} = \{\langle res_1, \ldots res_m \rangle, res\}$

Figure 4.4. Knowledge gained during execution R of protocol $\mathcal{Q}_0$

As a final note before presenting our security analysis, our analysis focuses on a specific adversarial model. Specifically, we assume the Dolev-Yao [81] adversarial model, in which an adversary can eavesdrop or modify any message. Furthermore, our analysis focuses on *rational* attacks. That is, we assume that RM, AS, and FMS, participate honestly unless they could benefit from deviating. In fact, as these principals have a vested interest in protecting the resource, we find no rational attacks by them, with the exception of violating the desired privacy guarantees. As such, our analysis assumes honest participation by these principals, except where noted. Instead, we focus on attacks in which an authorized user attempts to exceed his or her privileges (*e.g.,* eavesdropping on another user reading a file), or external adversaries attempting to gain illicit access to the system. In either case, the adversary would benefit by deviating, so we find these attacks rational and focus on them in our analysis. Lastly, for Protocol $\mathcal{Q}_0$, we exclude the feature portal from our analysis and consider it to be a trusted third party.

Lemma 4.4.1 *Replay attacks by an external adversary are detectable except with negligible probability.*

Proof Assume that z is nonce that is n bits in length. Then the probability that two users select the same z in two separate runs of the protocol is $1/2^n$. Furthermore,

σ_{ua} also depends on the timestamp ts. If we let m denote the number of protocol runs that can be initiated within $ts \pm \delta$, where δ denotes the maximum time for which the timestamp ts would be valid, then the probability that two randomly selected nonces are the same would be $m/2^n$. For moderate values of n, this probability is negligible. Thus, by keeping a log of the most recent m nonces used, RM would detect the replay with near certain probability.

Furthermore, even if RM fails to detect the replay, the only valid strategy for an adversary $\mathcal{A}$ would be to replay the exact messages. That is, as knowledge gained by $\mathcal{A}$ is $\phi_{\mathcal{A},R} \not\models id_{user}, cred_{role}, cred_{feat}$, $\mathcal{A}$ cannot forge a message $\widehat{\sigma_{ua}}$ for which $auth_{id}$ would be successful, other than the original σ_{ua}. In such a case, the effect of the replay would be contingent upon act. If act involves a modification, the replay would simply repeat the modification. Again, though, this succeeds with only negligible probability. However, if act is a read, the attack cannot succeed at all, as $\phi_{\mathcal{A},R} \not\models K_r$. Thus, $\mathcal{A}$ cannot decrypt the object and the attack fails. □

Lemma 4.4.2 *Tampering by an external adversary fails except with negligible probability.*

Proof This property follows from the fact that (Gen, Enc, Dec) is IND-CPA-secure and $(Gen, Sign, Ver)$ is CMA-secure. As such, $\mathcal{A}$ cannot forge $\widehat{\sigma_i}$ for any of the encrypted messages σ_i or $\widehat{\gamma_j}$ for any of the MACs γ_j. Thus, any attempt at tampering with the messages would be detected by the honest recipient. At best, $\mathcal{A}$ could induce a denial-of-service by modifying, for instance, any of the res_i values, causing γ_f to fail verification. However, the adversary cannot forge any message that would be accepted as legitimate, except with negligible probability. □

Theorem 4.4.1 *Protocol $\mathcal{Q}_0$ is secure under the Dolev-Yao adversarial model.*

Proof Follows from the preceding two lemmas. □

Lemma 4.4.3 *Protocol $\mathcal{Q}_0$ preserves user privacy from the RM.*

Proof This follows from the fact that $\phi_{RM} \not\models id_{user}$. That is, RM is able to receive an authenticated evaluation of the proximity constraints without having to know the identity of the requesting user. □

Lemma 4.4.4 *Protocol $\mathcal{Q}_0$ prevents replay by authorized users.*

Proof The primary concern here is that a user may exploit the asynchronous, distributed nature of the protocol to exercise a right when the proximity constraint no longer holds. However, $\phi_{RM,R} \models ts$, ensuring RM has the ability to validate that the timestamp claimed is reasonably accurate. Furthermore $\phi_{AS,R} \models ts$ and $\phi_{FMS,R} \models ts$, ensuring all principals see the same timestamp. Finally, γ_r is derived from ts, which allows AS and (indirectly) FMS to validate that the timestamp matched the time of request, as checked by RM. Thus, the timestamp and the MACs ensure that messages exchanged match the time used to evaluate the proximity constraint. □

Theorem 4.4.2 *Protocol $\mathcal{Q}_0$ provides strong authentication of user credentials and proximity claims.*

Proof Note that a successful execution R of $\mathcal{Q}_0$ requires $\phi_{AS,R} \models auth_{id}$. At the same time, $\phi_{AS,R} \models auth_{id} \supset \theta_U \models cred_{role}$. Thus, strong user authentication is provided by the assumptions regarding the *Auth* primitive. Additionally, $\phi_{FMS,R} \models auth_{feat} \supset \theta_U \models cred_{feat}$. As this credential is created by the centralized portal, which is beyond the scope of attack by a PPT adversary $\mathcal{A}$, the protocol integrates strong proximity authentication. □

Protocol $\mathcal{Q}_0$ is appropriate for proximity constraints defined for a centralized application. That is, social proximity assumes the use of a social network application with a global view. Similarly, cyber proximity is generally built on the assumption of a centralized service, such as telephony, though peer-to-peer designs (*e.g.,* Skype) also exist; in the latter case, Protocol $\mathcal{Q}_0$ would be inappropriate. Temporal proximity can be ensured assuming actions can be synchronized within the system. Applying Protocol $\mathcal{Q}_0$ for geographical proximity would be very challenging, as pervasive location monitoring is difficult.

Finally, note that there is a possibility for performance optimizations in certain deployments of Protocol $\mathcal{Q}_0$. Specifically, if AS and FMS are hosted on the same machine, the encryption of σ_{af} and the MAC γ_f are extraneous. That is, if the data exchanged by these two principals occurs over a secure channel in which eavesdropping is not possible, then the additional cryptographic protections are not necessary.

Figure 4.5 shows Protocol $\mathcal{Q}_1$, which extends the previous protocol to facilitate communication for the architecture in Figure 4.1(b). Most of the protocol is identical, with the exception being the generation of $cred_{feat}$. In Protocol $\mathcal{Q}_0$, this credential is assumed to be generated by the portal and is tangential to the protocol. In Protocol $\mathcal{Q}_1$, however, the user must explicitly retrieve the credential, which binds the user to a role and feature at a given timestamp $\widehat{ts}$. In this scenario, $Bind$ is assumed to be a computationally binding, perfectly hiding commitment scheme, while $GenValidation$ is a non-interactive proof. For instance, the two primitives may constitute a zero-knowledge proof-of-knowledge. The key is that U must not be able to forge such a credential in polynomial time. The remainder of the protocol is identical, with the exception that the proof needs to be forwarded to FMS, which verifies the credential.

The analysis of Protocol $\mathcal{Q}_1$ is virtually identical to Protocol $\mathcal{Q}_0$. Figure 4.6 shows the PCL knowledge sets for Protocol $\mathcal{Q}_1$. In this scenario, we are assuming the simplest case, in which the exchange between U and FM occurs in an insecure manner. For instance, this data may be exchanged over unencrypted Wi-Fi. Formally, this means $\phi_{\mathcal{A},R} \models cred_{feat}, valid_{feat}, \widehat{ts}$. However, the following lemmas show that this is not a security threat. Alternatively, this point could be made moot by using a secure channel between U and FM.

Lemma 4.4.5 *Protocol $\mathcal{Q}_1$ remains secure against a PPT adversary under the Dolev-Yao model.*

Proof As shown in Figure 4.6, $\phi_{\mathcal{A},R} \models cred_{feat}, valid_{feat}, \widehat{ts}$. However, as $Bind$ is assumed to be perfectly hiding, $\phi_{\mathcal{A},R} \not\models cred_{role}$. Furthermore, as $Bind$ is computationally binding, $\mathcal{A}$ could not forge the credential within polynomial time, even with

Protocol $\mathcal{Q}_1$ – feature monitor with no direct communication

1) $[U \rightarrow FM]\ id_{user}, role$
   $[FM]\ cred_{feat} := \text{Bind}(id_{user}, role, feat)$
   $[FM]\ valid_{feat} := \text{GenValidation}(cred_{feat})$
2) $[FM \rightarrow U]\ cred_{feat}, valid_{feat}, \widehat{ts}$
   $[U]\ \sigma_{uf} \leftarrow \text{Enc}_{pk(FMS)}(valid_{feat}, \widehat{ts})$
3) $[U \rightarrow RM]\ \sigma_{ur}, \sigma_{ua}, \sigma_{kr}, \sigma_{uf}$
4) $[RM \rightarrow AS]\ \sigma_{ra}, \sigma_{uf}$
5) $[AS \rightarrow FMS]\ \sigma_{af}, \sigma_{uf}$
   $[FMS]\ (valid_{feat}, \widehat{ts}) := \text{Dec}_{sk(SMF)}(\sigma_{uf})$
6) $[FMS \rightarrow AS]\ \langle res_1, \ldots, res_m \rangle, \gamma_f$
7) $[AS \rightarrow RM]\ res, \gamma_a$
8) $[RM \rightarrow U]\ \sigma_{res}$

Figure 4.5. Protocol for architecture in figure 4.1(b)

$\theta_U = \{obj, act, role, id_{user}, cred_{role}, ts\}$
$\theta_{RM} = \emptyset$
$\theta_{AS} = \{Auth, FindPolicies\}$
$\theta_{FMS} = \{Auth, (\mathcal{U} \times \mathcal{S})\}$
$\phi_{U,R} = \{retval, z, K_r, cred_{feat}, valid_{feat}, \widehat{ts}\}$
$\phi_{RM,R} = \{K_r, obj, act, z, res, retval, ts\}$
$\phi_{AS,R} = \{obj, act, role, id_{user}, cred_{role}, cred_{feat}, z, ts, auth_{id}, \langle res_1, \ldots, res_m \rangle\}$
$\phi_{FMS,R} = \{id_{user}, \langle pol_1.pt, \ldots pol_m.pt \rangle, cred_{feat}, valid_{feat}, \widehat{ts}, z, ts\}$
$\phi_{\mathcal{A},R} = \{\langle res_1, \ldots res_m \rangle, res, cred_{feat}, valid_{feat}, \widehat{ts}\}$

Figure 4.6. Knowledge gained during execution R of protocol $\mathcal{Q}_1$

the knowledge in $\phi_{\mathcal{A},R}$. Thus, $\mathcal{A}$ cannot forge σ_{ua} with the stolen credential in a manner that would be accepted by AS, except with negligible probability as described previously. Therefore, the stolen credential cannot be used to create unauthorized access. □

Lemma 4.4.6 *Protocol $\mathcal{Q}_1$ retains the privacy protection against RM as Protocol $\mathcal{Q}_0$.*

Protocol $\mathcal{Q}_2$ – feature monitor protocol

1) $[U \rightarrow FM]$ $id_{user}, role$
   $[FM]$ $cred_{feat} :=$ Bind$(user, role, feat)$
   $[FM]$ $valid_{feat} :=$ GenValidation$(cred_{feat})$
2) $[FM \rightarrow FMS]$ $valid_{feat}$
3) $[FM \rightarrow U]$ $cred_{feat}$
4) $[U \rightarrow RM]$ $\sigma_{ur}, \sigma_{ua}, \sigma_{kr}$
5) $[RM \rightarrow AS]$ σ_{ra}
6) $[AS \rightarrow FMS]$ σ_{af}
7) $[FMS \rightarrow AS]$ $\langle res_1, \ldots, res_m \rangle, \gamma_f$
8) $[AS \rightarrow RM]$ res, γ_a
9) $[RM \rightarrow U]$ σ_{res}

Figure 4.7. Protocol for architecture in figure 4.1(c)

Proof This follows from the fact that $\phi_{RM,A}$ is identical for the two protocols. Thus, Protocol $\mathcal{Q}_1$ continues to protect user privacy. □

Lemma 4.4.7 *Protocol $\mathcal{Q}_1$ continues to provide strong feature authentication for authorized users.*

Proof The computationally binding nature of *Bind* prevents forgery of $cred_{feat}$ by U except with negligible probability. Thus, FMS, when validating the credential, has probabilistic assurance that the credential has not been forged. Furthermore, as $\phi_{FMS,R} \models \widehat{ts}, ts$, FMS can determine that the request was made within an acceptable time frame of the feature credential creation. □

Finally, Figure 4.7 shows Protocol $\mathcal{Q}_2$, which defines an extension for the architecture in Figure 4.1(c). As in $\mathcal{Q}_1$, FM is responsible for issuing a computationally binding credential. However, the binding routine in this protocol is more flexible. That is, in Protocol $\mathcal{Q}_1$, $valid_{feat}$ must be implemented as a non-interactive technique, such as a cryptographically signed certificate. In contrast, in Protocol $\mathcal{Q}_2$, FM pushes the credential validation information to FMS. For instance, FM could

generate a Pedersen commitment [82], sending the commitment to FMS while sending the data to open the commitment to the user. Note, though, that this protocol includes the same knowledge sets as $\mathcal{Q}_1$, with the exception that $\phi_{U,R} \not\models valid_{feat}$. As such, the security analysis remains unchanged.

4.5 Conclusion

In this chapter we have explored various notions of proximity. Specifically, we have discussed five types of proximity: geographical, attribute-based, cyber, social and temporal. We have developed a formal model of proximity that is generic enough to specify all these types of proximity. We have presented theoretical results illustrating the challenges inherent in implementing a proximity-based RBAC system, and we have also described approaches to overcome these difficulties. We have presented three generic enforcement architectures and provided protocol templates for enforcing such systems, formalized these protocols using PCL and proved security properties of these protocols. In summary, we argue that it is feasible to deploy a practical proximity-based RBAC system for a variety of contextual factors.

5 DEFENSE AGAINST CODE REUSE ATTACKS

Return Oriented Programming (ROP) attacks are an advanced form of buffer overflow attacks [83] that reuse existing executable code towards malicious purposes. While earlier exploits involved the injection of malicious code [83], the recent trend has been to reuse executable code that already exists, primarily in the application binary and shared libraries such as *libc*. These code reuse attacks can bypass traditional defenses against code injection attacks such as $W \oplus X$ protection [37] that prevents execution of arbitrary code that is injected into the memory. In a basic code reuse attack, for instance return-into-*libc* attack [84, 85], a buffer overflow corrupts the return address to jump to a *libc* function, such as `system`. This type of attack then evolved into a more generic ROP attack [86]. In ROP, the attacker identifies small sequences of binary instructions, called *gadgets*, 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 [87].

In considering a new defensive technique, we start with two observations. First, the main shortcoming of earlier randomization-based techniques was insufficient entropy, thus making brute-force attacks feasible. Second, executable code can naturally be broken into many *function blocks* that can potentially be shuffled. Consequently, the amount of possible randomization generated can be significantly increased by permuting these code blocks within the executable. For instance, if an application has 500 function blocks, there are $500! \approx 2^{3767}$ possible permutations of these function blocks which significantly increases the brute force effort required from an attacker.

We are not the only researchers to have investigated software diversity for ROP attack mitigation. As discussed in section 2.3, the other approaches suffer from one or more of the following limitations. First, the software diversification is not done

frequently enough. Second, some of the existing defenses require the source code or other additional information that is not usually available. Third, the randomization is not fine grained enough leaving large code chunks unrandomized. Fourth, significant runtime overhead is incurred throughout the runtime of the application by introducing additional data structures. Marlin addresses these limitations and provides a strong and efficient defense technique against ROP attacks.

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. We have evaluated the time to randomize compiled binaries on a selection of commonly used applications and Linux `coreutils`, showing that the performance penalty for Marlin is reasonable in the average case. Thus, our work demonstrates that, although Marlin imposes certain performance costs, its success in thwarting ROP attacks makes this a feasible approach for systems that prioritize execution integrity over optimal performance. In section 5.2 we describe techniques for minimizing this performance impact. For instance, performing the randomization during offline pre-processing significantly reduces the startup costs.

The remainder of this chapter is structured as follows. We start by discussing code-reuse attacks techniques in section 5.1. In section 5.2, we describe our approach in more detail, including optimization techniques to reduce overhead. Section 5.3 discusses the implementation details of Marlin. Section 5.4 shows the results of various experiments that were performed to evaluate our approach. In Section 5.5, we discuss the merits and limitations of Marlin. We then conclude in Section 5.6.

5.1 Background and Related Work

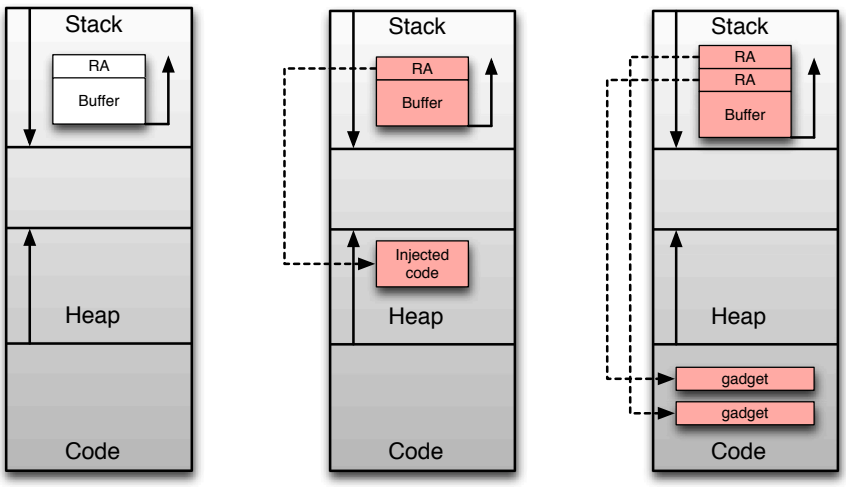
The focus of our work is on ROP attacks, which are a special case of code-reuse attacks that leverage existing code in the application binary to execute arbitrary instructions. In this section, we start with a brief summary of these attack techniques.

Figure 5.1 shows the evolution of buffer overflow attacks. After describing the attack techniques, we summarize critical factors of code-reuse attacks.

5.1.1 Return-oriented Programming

Return-oriented programming (ROP) is an exploit technique that has evolved from stack-based buffer overflows as shown in Figure 5.1. 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 [88] and unaligned CISC architectures [86]. ROP techniques can be used to create rootkits [89], can inject code into Harvard architectures [90], and have been used to perform privilege escalation in Android [91]. Initiating a ROP attack is made even easier by the availability of architecture-independent algorithms to automate gadget creation [92].

While researchers were exploring defenses against return-oriented attacks, similar techniques can manipulate other instructions, such as `jmp` and their variants [87, 93, 94]. 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. Although these attacks do not require `ret` instructions, researchers traditionally have included them in the category of ROP attacks.



(a) Original layout (b) Injected code (c) ROP attack

Figure 5.1. Evolution of buffer overflow attacks

5.1.2 Enabling Factors for Code-reuse Attacks

Based on our survey of ROP attacks and defenses, we have identified distinct characteristics and requirements for a successful exploit. The fundamental assumption and enabling factor for such attacks is as follows:

The relative offsets of instructions within the application's code are constant. That is, if an attacker knows any symbol's address in the application code, then the location of all gadgets and symbols in application's codebase is deterministic.

We argue that a defensive technique that undermines these invariants will present a robust protection mechanism against these threats.

5.2 Marlin Defense Technique

Code-reuse attacks make certain assumptions (as discussed in section 5.1.2) about the address layout of application's executable code. Marlin's randomization technique aims at breaking these assumptions by shuffling the code blocks in the binary's `.text` section with *every* execution of this binary. This significantly increases the difficulty

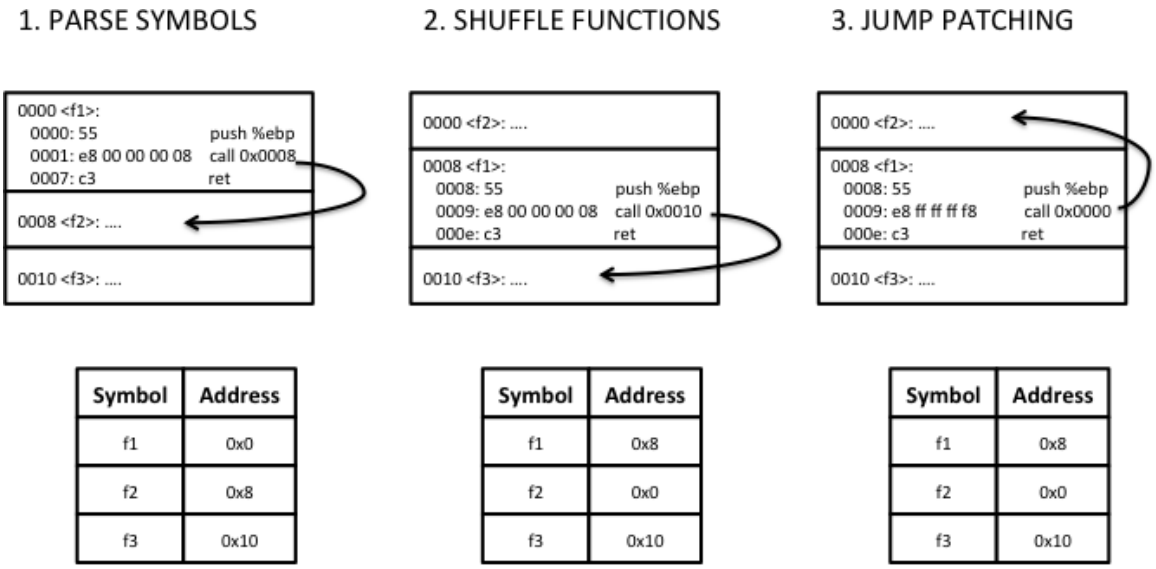
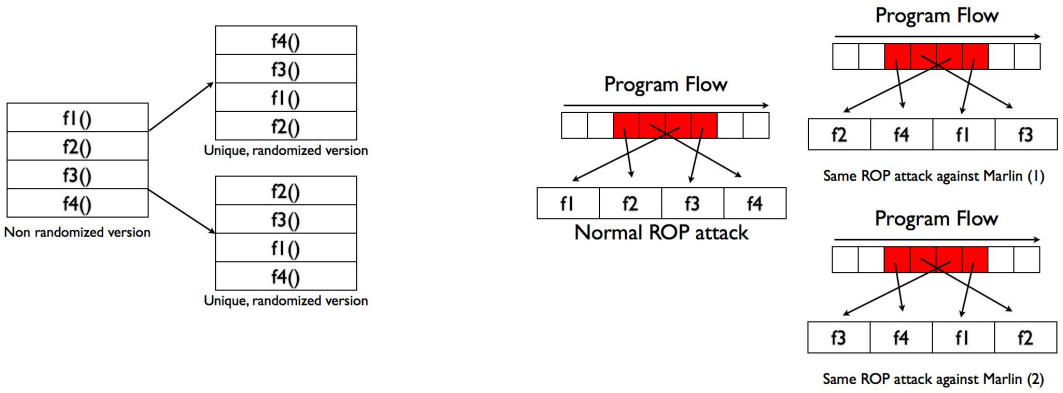


Figure 5.2. Processing steps in Marlin

of such attacks since the attacker would need to guess the exact permutation being used in the current process execution. This shuffling is performed at the granularity of function blocks as discussed in section 5.2.2. The various steps involved in Marlin processing are shown in Figure 5.2. Marlin is integrated into a modified bash shell that randomizes the target application just before the control is passed over to this application for execution. Thus, every execution of the program results in a different process memory image as illustrated in Figure 5.3(a). Figure 5.3(b) illustrates how shuffling the code results in a sequence of gadgets that is not intended by the attacker. We now present Marlin technique in detail.

5.2.1 Attack Assumptions

We start by describing the basic assumptions for a ROP attack scenario. The vulnerable application may have a buffer overflow or heap overflow vulnerability that can be leveraged by an attacker to inject an exploit payload. The system is assumed to be protected using write or execute only policy ($W \oplus X$) and the attacker can not



(a) Unique output with every run (b) Mitigation of ROP attack

Figure 5.3. Effect of function block randomization

inject arbitrary executable code in the stack or the heap. The attacker is assumed to have access to the target binary that has not yet undergone Marlin processing. The attacker is also assumed to be aware of the functionality of Marlin. However, the attacker can not examine the memory dump of the running process and is unaware of how exactly the code is randomized for the currently executing process image. Our approach protects against both remote and local exploits as long as the attacker is not able to examine the memory of the target process. For instance, in this threat model, a local attacker can not attach a debugger to a process that is running as root and obtain its memory dump.

5.2.2 Granularity of Randomization

Code can be randomized at various levels of granularity such as instruction level, basic block level, function level, segment level or just the base address. Choosing the right granularity of randomization is a tradeoff between effectiveness (measure of security offered) and efficiency (measure of overhead incurred) of the the resulting defense scheme. While randomizing at finer granularity (such as basic block or instruction level) provides higher entropy, it may also incur higher overhead as it breaks

the principle of locality. That is, the basic blocks that comprise a function might be moved to different pages and the system would have to load multiple memory pages to execute a single function. Randomization at basic block also involves handling many more types of jumps and calls which require precise control flow graph information. Such precise control flow information typically cannot be completely extracted from an executable. This makes basic block based randomization more difficult to handle in the absence of complete control flow graph information.

On the other hand, randomization at the function level granularity eases the handling of several jumps and calls. As the function body remains intact during the shuffling phase, relative jumps are not affected as their target lies within the same function. Same holds true for certain computed jumps. This avoids patching the target address for near jumps that occur within a function body. Also, by keeping all the basic blocks of a function body together, the overhead of loading multiple pages per function can be avoided. For these reasons, we chose to implement randomization at the function level granularity in Marlin. We have shown later in the evaluation section that even with this coarse granularity, it offers strong protection against brute force attack. We now discuss the various steps involved in application code randomization.

5.2.3 Preprocessing Phase

As mentioned above, Marlin randomizes the application binary at the granularity of function blocks. This requires identifying the function blocks in the application binary. In preprocessing phase, the ELF binary is parsed to extract the function symbols and associated information such as start address of the function and length of the function block. However, traditional binaries are typically stripped binaries and do not contain symbol information. In such cases, we first restore the symbol information using an external tool, *Unstrip* [95]. Once the symbol information is

restored and identified, we proceed on to the next stage of Marlin processing that randomizes the application binary.

5.2.4 Randomization Algorithm

Once the function symbols have been identified, Marlin generates a random permutation of this set of symbols. The function blocks are then shuffled around according to this random permutation. Shuffling the function blocks in an application binary changes the relative offsets between instructions that may affect various jump and call instructions. The target destination for these jumps/calls can be specified either as an absolute address or as a relative offset. Relative jumps increment or decrement the program counter by a constant value as opposed to absolute jump that directly jump to a fixed address. When the function blocks are randomized, these jumps will no longer point to the desired location and must be 'fixed' to point to the proper locations. We achieve this by performing *jump patching*.

The randomization algorithm described in Algorithm 2 involves two stages. In the first stage, the function blocks are shuffled according to a certain random permutation. During this shuffling, we keep a record of the original address of the function and also the new address where the function will reside after the binary has been completely randomized. This information is stored in a *jump patching table*. Note that this jump patching table is discarded before the application is given control, thus preventing attacker from utilizing this information to de-randomize the memory layout. In the second stage, the actual jump patching is done where the jump patching table is examined for every jump that needs to be patched. Whenever a relative jump is encountered, the algorithm executes `PatchRelativeJump()` method to redirect the jump to the correct address in the binary. `PatchRelativeJump()` method takes the current address of the jump and the address of the jump destination to determine the new offset and patch the jump target. The second case is the computed jumps where the contents of a register specify the absolute address of the destination, for

Algorithm 2: Code randomization algorithm

Input : Original program, P

Output: Randomized program, P_R

L = All symbols in P

F = A list of forbidden symbols that should not be shuffled

$L = L - F$

O_L = Ordered sequence of symbols in L

$S.Addr_P$ = Address of symbol S in program P

$J.Addr_P$ = Address of jump instruction J in program P

$J.Dest_P$ = Destination address of jump J in program P

$J.Sym$ = Symbol that J is jumping into

```
/* Permutation stage */
```

for *Every symbol* $S \in L$ **do**

- R = Randomly select another symbol in L
- Swap S and R in O_L

P_R = Permuted program according to symbol order in O_L

```
/* Jump patching stage */
```

for *Every symbol* $S \in L$ **do**

- **for** *Every jump* $J \in S$ **do**
  - **if** *J is a relative jump to within S* **then**
    - `/* No action needed */`
  - **else if** *J is a relative jump to outside S* **then**
    - $J.Dest_{P_R} =$
    - $J.Dest_P + (J.Sym.Addr_{P_R} - J.Sym.Addr_P) - (S.Addr_{P_R} - S.Addr_P)$
    - PatchRelativeJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)
  - **else if** *J is an absolute jump* **then**
    - PatchAbsoluteJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)

example call to function pointers. We handle these cases by doing a backward analysis and fixing the instruction where the function address is being loaded into a register. If the destination address is obtained from `.data` section (for example, in case of global function pointers), then we patch the `.data` section with the new value. This processing is done in `PatchAbsoluteJump()` method shown in Algorithm 2.

The run-time shuffling of the function blocks prevents multiple instances of the same program from having the same address layout. Thus, to defeat Marlin, an attacker would need to dynamically construct a new exploit *for every instance of every application* which is not possible since the randomized layout is not accessible to attacker. We now discuss the security guarantees offered by Marlin.

5.2.5 Security Evaluation

We now show that our randomization technique significantly increases the brute force effort required to attack the system. In a brute force attack, the attacker will randomly assume a memory layout and craft exploit payload according to that address layout. A failed attempt will usually cause a segmentation fault due to illegal instruction and the crashed process or thread will need to be restarted. We now compute the average number of attempts required by an attacker to succeed. A successful attack is assumed to be equivalent to guessing the correct permutation used for randomization.

In the discussion that follows, let n denote the number of symbols (excluding forbidden symbols) in an application binary. The total number of possible permutations that can be generated for this application is $N = n!$. Let $P(k)$ denote the probability that the attack is successful for the first time at the k^{th} attempt. Let X be a random variable denoting the number of brute force attempts after which the attack is successful for the first time (that is, the attacker guesses the correct permutation). We will now estimate the *average* value of X. We consider the following two cases.

Case 1: A failed attempt crashes the process and causes it to be restarted.

In this event, the process will be restarted with a new randomization. The subsequent brute force attempts by an attacker will be independent since he would learn nothing from the past failed attempts. That is, the probability of success at k^{th} attempt is constant and independent of k. Let $p = \frac{1}{N}$ denote the probability of success at any attempt. Then, the average number of attempts before the attack is successful for the first time is

$$\mathrm{E}[X] = (p * 1) + (1 - p) * (1 + \mathrm{E}[X]) = \frac{1}{p}$$
$$\Rightarrow \mathrm{E}[X] = n!$$

Thus, the attacker would have to make an average $n!$ number of attempts to correctly guess the randomized layout and launch a successful ROP attack.

Case 2: A failed attempt crashes a thread of the process and causes only that thread to be restarted.

In this event, since the process is still executing, the memory layout will remain same. Every failed attempt will eliminate one permutation. The probability that first success is achieved at k^{th} attempt is

$$P(k) = \left(\prod_{i=1}^{k-1} \frac{N - i}{N - i + 1} \right) * \frac{1}{N - k + 1} = \frac{1}{N}$$

The average number of attempts before first success can be computed as

$$\mathrm{E}[X] = \sum_{x=1}^{N} x * P(x) = \sum_{x=1}^{N} x * \frac{1}{N} = \frac{N+1}{2}$$
$$\Rightarrow \mathrm{E}[X] = \frac{n! + 1}{2}$$

So, the attacker will need an average $\frac{n!}{2}$ number of brute attempts to correctly guess the randomization and launch successful ROP attack. Given enough time and resources, the attacker can try all possible permutations one after the other and will require at most $n!$ attempts for a successful brute force attack.

As an example, to launch a successful ROP attack against an application with 500 symbols that is protected using Marlin, an average $500! = 2^{3767}$ number of attempts

will be required for the first case. This is clearly computationally infeasible. A more extensive evaluation performed using `coreutils` applications is presented later in Section 5.4 that demonstrates the effectiveness of our technique.

5.2.6 Discussion

Having described our randomization techniques above, it is necessary to offer a few words about how Marlin applies them while addressing specific implementation challenges that have been identified [4] in regard to memory image randomization. Against ROP attacks, randomization is, by far, the more effective technique. By significantly increasing the entropy of the application image, randomization creates negligible probability that an adversary can craft a chain of gadgets for short-lived applications, as every new process will have a different configuration of function blocks. Specifically, the large number of possible permutations significantly increases the number of attempts needed for a single ROP gadget chain to work.

Shacham *et al.* [4] correctly point out that full randomization eliminates sharing memory pages between processes. For strong security guarantees, *eliminating sharing is actually desirable*. That is, for some critical applications, it is more important to guarantee integrity than optimal performance. However, in other cases, such strong security guarantees are not required. To accommodate a wide range of trade-offs, several approaches are possible. First, an executable could be marked as *critical*, which would then be fully randomized. Next, *normal* applications would first detect if another instance of the same executable is already running. If so, the new process would share read-only access to the shuffled code image. Such options can be implemented using flags that get passed to Marlin.

5.2.7 Optimization Techniques

A straightforward performance optimization for Marlin would be to perform the pre-processing for jump patching only once for each application and store the result

in a database maintained by the system. The jump patching algorithm can reuse the information about function blocks from this database in subsequent executions. The database would only need to be updated when the application code changes.

The impact of the code randomization can be reduced 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 binary is executed, the custom shell sends a signal to a trusted daemon process that runs with low priority and returns the next permutation. The application's function blocks are then shuffled accordingly.

5.3 Implementation Details

We have implemented a Marlin prototype that can operate on any ELF binary without requiring its source code. The implementation was done for 32-bit x86 architecture on a system running Ubuntu operating system. Implementation of Marlin involved two major components. First part consisted of randomizing the executable code and generating the randomized binary. The second part dealt with integrating this into an existing system such that binary randomization occurs seamlessly with every execution. We discuss the details of Marlin implementation below.

5.3.1 Code Randomization

Randomizing an application's executable code segment consists of two stages. First is the preprocessing stage that can be done just once per binary and is independent of subsequent executions. This stage involves disassembling a binary and extracting information about the function blocks and also the control flow. The second stage is the actual randomization stage when the function blocks are shuffled and the jump/call targets are patched. We now discuss each of this in further detail.

Preprocessing Stage

Before we randomize the binary, we need to identify the function blocks. We do this by disassembling the binary using `objdump` disassembler and then parsing the dissembler output to extract the function symbols and the relevant information. For each function symbol, we gather information about its location in the executable, the length of the function block and the information on any jumps or calls originating from this function. This information is collected for functions in the PLT table as well in addition to the application defined functions.

While we use `objdump` disassembler, other commercial options such as IDAPro can be used to obtain more accurate disassembly. Also, several production level binaries are available only as stripped binaries, that is the symbol information has been removed from them. We restore the symbol information using `Unstrip` utility [95] before disassembling it using `objdump`.

Randomization Stage

In this stage, the actual shuffling of the function blocks is performed. The first step is to generate a random permutation of symbols and shuffle the list of symbols to obtain a new order of symbols. The new binary is re-written according to this new symbol order. In our preliminary implementation [96], we did not shuffle certain symbols such as `_start` that were referred to as *forbidden symbols*. Our revised implementation no longer has this limitation and all the symbols within `.text` section are now randomized, including `_start` symbol. This `_start` symbol is the first instruction that executes after the binary is loaded into the memory by the ELF loader. This entry address is stored in ELF header of the binary. Once the application is randomized, we patch the ELF header with the new entry address which is the new location of `_start` symbol.

Fixing Jumps and Calls

The jump and call patching is performed in the same pass when the new randomized binary is written. This is done by using the patch list information that is generated during the preprocessing stage. For each call that needs to be patched, the patch information consists of the name of the parent symbol, the name of symbol being patched to and the offset from the beginning of the parent symbol where the patching needs to be done.

The calls and jumps can be of the following types:

- Call instructions
  - Call to an application defined function: In normal function calls ($call < f1 >$), the target address of the callee function is specified as relative address offset from the address of the call instruction. We fix this target address in the call patching phase using the patch information collected during the preprocessing phase.
  - Call to a dynamically linked function: Functions in dynamically linked libraries that are called in the application's code appear in the PLT section of the application's code. Calls to these linked functions ($call < f2@plt >$) are also specified as relative offset from the address of the call instruction to the function's PLT entry. These targets are also fixed in the call patching stage by correcting the relative offset.
  - Call to a function pointer: Call to function pointers are handled as indirect calls, that is the absolute address of call target is loaded into a register, say $\%eax$, and then the call is made as $call * \%eax$. To fix these types of calls, the absolute address of the callee should be patched at the instruction that loads its address into the $\%eax$ register. This is done by doing a backward analysis starting from an indirect call instruction and tracing backwards until we reach the instruction where the value of function pointer is loaded. In case of global function pointers, the address of function is stored in the

data section and eventually loaded from this data section into a register. In these cases, the data section is patched with the corrected function address after shuffling.

- Jump Instructions: In x86 architecture, jumps can be either conditional jumps or unconditional jumps. Conditional jumps are near jumps while unconditional jumps can be either near or far jumps. We don't need to patch the near jumps as they are within the same function body and specified using the function offset. However, unconditional far jumps transfer program control to the target address without a return. For example, this can happen in the case of goto statement where the jump specifies an absolute address . If the jump destination is outside the application's code, for example a shared library, then this does not need patching. However, if the destination of a far jump is within the application code, then this will need to be patched. We patch certain far jumps, for instance the jump tables that are created due to some `switch-case` blocks. These jump tables are stored in `.rodata` section of the code. We patch the jump table in this `.rodata` with the new jump targets after randomizing the code.

5.3.2 System Integration

Software diversification can be applied at various stages in an application's lifecycle ranging from compile-time diversification to runtime-diversification. In Marlin, our goal is to randomize the target binary with each execution. That is, we want to invoke the Marlin functionality whenever the target binary is executed. We considered several approaches to achieve this as discussed below.

First approach that we considered was modifying the dynamic loader (`ld`). In this approach, the application code that is mapped in the memory by `mmap` call will be randomized just before the control is passed to the target binary. This has the advantage that all the transformations are done in the memory and they are done immediately before the control flow jumps to application's `_start` symbol. This en-

sures that every application is randomized with each execution and no intermediate files are generated that can be potentially exploited by the attacker. However, this approach incurs several complications and redundant processing. For example, the loader library (`ld-linux.so`) is self contained and does not use any shared library which makes it difficult to integrate Marlin's code into the loader. More importantly, the work that the loader has done to load the normal application code (resolving references etc) is wasted effort since the functionality needs to be re-executed after randomizing the code again. For these reasons, we decided against using this approach.

The second approach that we considered was modifying the `execve` system call such that it first executes Marlin to randomize the target executable and then executes the randomized code. However, since `execve` is used by almost every execution in the system including the kernel code, modifying this function can introduce a lot of instability into the system, especially during the testing phase. Also, to prevent the recursive invocations of Marlin onto itself, one would need to modify the `execve` definition which is not a good solution since `execve` is called at several places in both user and kernel code. Thus, we decided against using this approach as well.

Finally, the third approach that we considered was a custom secure shell approach. In this approach, we modified the normal shell code to create a secure shell that would randomize the target binary before executing it. This is the approach that we adopted in implementing the Marlin prototype. We modified the source code for bash shell, specifically the `shell_execve` function that is responsible for making a call to `execve` method. We created a hook just before the `execve` call to randomize the code for the target binary. This approach has the advantage that it allows us to test our system without interfering with the existing system functionality. In the deployment of the production version, one can easily replace the normal bash shell with our secure shell to ensure that all the executions invoked by this shell are randomized.

Further, we implemented a whitelist that allows for selectively randomizing only certain application binaries. For example, one may wish to randomize only those

applications that have user interaction, that is they accept input from a user (can be a remote user). Our implementation supports specifying the whitelist entries in three different ways. First, the entry can be an absolute path of a directory in which case all the files contained in this directory and its sub-folders will be randomized. Second, the entry can be specified as the absolute path of an application in which case this application is randomized whenever it is executed using secure shell. Finally, the entry can be specified just using the name of the executable in which case any executable with the specified name, irrespective of its path, will be randomized before execution. This whitelist is protected and can only be modified by the superuser.

5.4 Evaluation

We now describe various experiments that were performed to evaluate *Marlin* technique. These experiments test the effectiveness of Marlin technique and also the performance overhead incurred due to randomization. The experiments were performed on a Linux virtual machine with 2 processor cores and 4GB RAM (host machine processor was Intel Core i5 2.4GHz with 6GB RAM). This VM had ASLR and $W \oplus X$ protection enabled while the experiments were being performed. We used `coreutils` binaries, some commonly used application binaries (see Table 5.1) and byte-unixbench [97] benchmarks to conduct various experiments. To launch attacks against Marlin-protected binary, we use ROPgadget (v3.3.3) [98], an attack tool that automatically creates exploit payload for ROP attacks by searching for gadgets in an application's executable section.

5.4.1 Effectiveness

First, we tested the effectiveness of Marlin using a test application that has a buffer overflow vulnerability. This application, `ndh_rop`, was included as a part of the ROPgadget test binaries. We used ROPgadget on this target application and found 162 unique gadgets. These were sufficient to craft a shell code exploit payload. When

Table 5.1.
List of applications used in evaluation

Application	**Version**
Apache	2.4.7
Bash	4.20
Brasero	3.11.0
Cups	1.7.0
Coreutils (105 applications)	8.22.1
Dhclient	4.2.5
Emacs	24.3
Gcc	4.8.1
Gedit	3.8.3
Ghex	3.8.1
Gimp	2.80
Git	1.8.5
Gnome-terminal	3.0.1
Gtkpod	2.1.4
Gzip	1.60
Lame	3.99.5
Make	3.81
Mono	–
Nano	2.3.2
OpenSSH	6.4
Qemu	1.7.0
Subversion	1.8.5
Tar	1.27
Vim	7.4
Vlc	2.1.2
Wine	1.1.27
Wireshark	1.11.2

this exploit payload was provided as an input to the unprotected binary, it gave us a shell. Next, we randomized this application using Marlin technique and tried to attack it using the same input payload. The attack did not succeed and failed to provide us with a shell.

This highlights the sensitivity of these attacks to slight changes in the address layout. ROP attacks operate under the strong assumption of a static address layout of executable code. In our threat model, the attacker only has access to the unprotected binary and is not aware of the exact permutation that has been used for randomization. So he can only run ROPgadget tool on the unprotected test application.

Brute Force Effort

In section 5.2.5, we computed the average number of attempts required to successfully attack a randomized binary. This brute force effort is approximately `n!` where `n` is the number of symbols in a binary. We performed an extensive evaluation of this using 131 ELF binaries corresponding to 105 `coreutils` applications and 26 commonly used applications. Figure 5.4 shows the CDF of number of symbols present in these applications. We noticed that around 97.7% of these applications have more than 80 symbols (indicating an effort of 80! attempts). We observed an average of 470 symbols and a median of 130 symbols present in these applications. Thus, the number of brute force attempts in a general case can be approximated to $130! \approx 2^{730}$ attempts which is quite significant. Also, on an average, we observed the time to compute one attack payload is 15.48 seconds.

It is interesting to note that the effectiveness of protection offered by Marlin depends on the modularity of the program. An application that has several function modules will be more secure against brute force attempts when protected with Marlin. If the entire code of an application is organized in few functions, then irrespective of the size of the binary, it will still be quite susceptible to brute force attacks since it would contain large chunks of unrandomized code. Randomizing at finer granu-

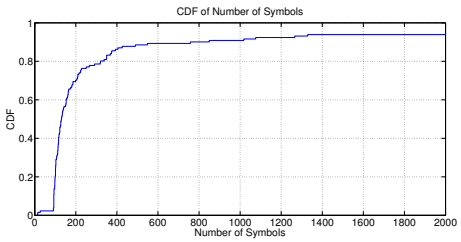


Figure 5.4. CDF for number of symbols

larity, for example at the granularity of basic blocks or instructions, will solve this issue. However, we believe that randomization breaks the locality principle and the randomized binary may suffer a performance hit. Thus, as a trade off, we chose to randomize at the granularity of function block.

Gadget Displacement

Next, we studied the entropy introduced by our randomization approach by measuring the gadget displacement. That is, we measured how many gadgets are moved due to randomization by Marlin technique. To measure this, we extend the ROPgadget tool to compare the original binary with the randomized binary and compute the number of unique gadgets that were found in former and are no longer present at the same address in latter. This experiment was also performed on the same set of 131 application binaries as used in section 5.4.1 with 20 iterations per binary.

We measure two types of gadget displacement. First, we measure the displacement of unique gadgets that are found by ROPgadget in the executable sections of the target binary. Note that these gadgets are not necessarily from `.text` section and may belong to other executable sections such as `.plt` section. In this case, we observed an average of 71.8% and a median of 72.5% gadgets were displaced in the randomized binaries. Since we randomize only `.text` section, the gadgets found in other executable sections were unmoved. Next, we restricted the search for unique gadgets to only `.text` section and measured the number of gadgets that were displaced by randomization. In this

case we observed an average of 99.78% gadget displacement (with median as 100% displacement). Thus, nearly all gadgets in the `.text` section are displaced.

We can conclude from above observations that randomizing at function level granularity leads to high gadget displacement which is quite effective against ROP attacks. This eliminates the need for randomizing at a more finer granularity such as basic blocks or instruction level.

5.4.2 Overhead Analysis

We evaluated the efficiency of Marlin by measuring two variables. First, we measured the processing cost incurred by Marlin while loading an application as discussed in section 5.4.2. Second, we measured the runtime overhead of the randomized binaries. This is discussed in more detail in section 5.4.2.

Marlin Processing Overhead

When an application is loaded, Marlin identifies the function blocks and records information about them (such as start address, length) that is used later in jump patching. This computation is independent of the individual randomizations. Next phase involves shuffling the function blocks and patching the jumps. Marlin processing cost is the combined overhead of these two phases. We measure Marlin processing overhead for the same set of 131 ELF binaries used in above experiments with 20 randomizations per binary.

Figure 5.5 shows the CDF of processing cost for Marlin for these 131 applications. We notice that 95% of these applications incurred less than 1.43 seconds processing time. This is quite reasonable since this is a one time overhead incurred only at the application load time. We observed that applications with larger number of symbols incur more processing overhead. For instance, the application gimp took significantly longer time to process (8.13 seconds). This is because it contained 10760 symbols in

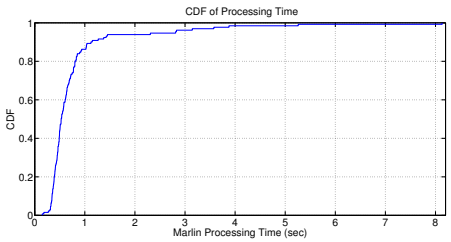


Figure 5.5. CDF for Marlin processing time

contrast to a median of 130 symbols by other applications. The average time taken by Marlin processing was 0.87 seconds with a median of 0.53 seconds.

Thus, we observe that the processing overhead due to Marlin is very minimal. Also, the performance hit is incurred only at the load time of the application. Once the application binary has been randomized, it executes like a normal application binary.

Runtime Overhead

We measured the runtime overhead of randomized binaries to see if shuffling the functions affects the execution time of a binary. For this purpose we use the byte-unixbench benchmarks. We used the execution time of un-randomized benchmarks as a baseline to compare with randomized benchmarks. We performed 20 randomizations per benchmark and took the average of these values. We observed that the execution time of the benchmarks was same before and after randomization and was not affected due to Marlin. This supports our initial hypothesis that the overhead is incurred only during the randomization phase of Marlin and after that the binary executes as a normal binary with no runtime overhead.

5.4.3 Comparison with Existing Defense Techniques

Tables 5.2 and 5.3 compare Marlin with other approaches with respect to the defense techniques, the properties, and the metrics. We compared these approaches

based on nine comparison dimensions. First, we looked at the types of code reuse attacks mitigated by these techniques. Marlin defends against both return-based and jump-based attacks, unlike [50,61] that can stop only return-based attacks. Next we look at the techniques adopted by these defense approaches. These techniques employ either some form of diversification or use execution monitoring. For diversification based approaches that use code randomization, the randomization can be performed at various granularities. While randomizing at finer granularity such as instruction level [40, 43] increases entropy, it decreases the runtime performance by breaking memory locality. Control flow integrity [58,60] and other approaches [50,52,61] that monitor runtime execution also incur significant runtime overhead. Marlin adopts a tradeoff approach by using a coarse-granularity of randomization that achieves high entropy with no runtime overhead.

These defense techniques can be applied at different stage of software cycle such as installation, loading, execution stage etc. Marlin is applied at application load time which provides the advantage of frequent randomization without affecting runtime performance. Techniques such as [40,50,52,58,60,61] that are deployed at execution stage incur a runtime overhead as shown in column 4 (in Table 5.2) and column 3 (in Table 5.3). Techniques that use runtime data structures also incur memory overhead in addition to runtime overhead. As shown in column 6 (in Table 5.3), [40,50,52] use runtime data structure, while Marlin technique does not require any such additional data structure. Thus, Marlin incurs no runtime or memory overhead.

Table 5.2.
Comparison with other defense techniques - Part I

Defense	**ROP types mitigated**	**Technique**	**Stage**	**Platform**
Marlin	ROP, JOP	Function shuffling	Load time	Linux (x86 ELF)
ILR [40]	ROP, JOP	Instruction shuffling	Installation, Execution	Linux (x86 ELF)
STIR [43]	ROP, JOP	Basic-block shuffling	Load time	Linux (x86 ELF), Windows (PE)
XIFER [42]	ROP, JOP	Code piece shuffling	Load time	Linux (x86 ELF), ARM
IPR [41]	ROP, JOP	Instruction reordering, Equiv. instruction substitution	Offline tool	Windows (PE)
DROP [61]	ROP only	Check gadget sequence length with threshold	Execution	Linux (x86 ELF)
ROPDefender [50]	ROP only	Instrumentation to check return address	Execution	Linux, Windows
ROPecker [52]	ROP, JOP	Check for long gadget chain	Installation, Execution	Linux (x86 ELF)
CCFIR [60]	ROP, JOP	Restrict jump target using white-list	Installation, Execution	Windows (x86 PE)
CFL [58]	ROP, JOP	Control flow locking	Compilation, Execution	Linux (x86 ELF)

Table 5.3.
Comparison with other defense techniques - Part II

Defense	**Benchmark**	**Runtime overhead**	**Space overhead**	**Gadget Displacement**	**Runtime data-structures**
Marlin	Linux coreutils, byte-unixbench, COTS	0%	0%	99.78%	No
ILR [40]	SPEC CPU 2006	13-16%	14MB - 264MB	99.96%	Yes (Fall-through map)
STIR [43]	SPEC CPU 2000, Linux coreutils, COTS	4.6% (SPEC) 0.3% (coreutils) 1.6% (overall)	73% (file size) 37% (process size)	99.99%	No
XIFER [42]	SPEC CPU 2006	1.2-5%	1.76% (file size), 5% (at runtime)	100%	No
IPR [41]	Wine test suite	0%	N/A	76.9%	No
DROP [61]	COTS	430%	N/A	N/A	No
ROPDefender [50]	SPEC CPU 2006	117% (SPECint) 49% (SPECfp)	Yes (Statistics not reported)	N/A	Yes (shadow stacks)
ROPecker [52]	SPEC CPU 2006, Bonnie++, Apache	2.6% (SPEC CPU) 1.5% (disk I/O) 0.08-9.72% (Apache)	210MB	N/A	Yes (instruction and gadget database)
CCFIR [60]	SPEC CPU 2000	3.6% (SPECint 2000) 0.59% (SPECfp 2000) 4.2% (SPECint 2006)	Yes (statistics not reported)	N/A	Yes (Springboard)
CFL [58]	SPEC CPU 2000, SPEC CPU 2006, COTS	0-21%	Yes (Statistics not reported)	N/A	No

These overheads for other approaches are indicated in columns 3 and 4 in Table 5.3. Some of the defense approaches such as [43, 50] use binary instrumentation to integrate their technique and this results in a increase in file size (see column 4 in Table 5.3). Marlin does not use any instrumentation and does not incur any space overhead.

We also compared Marlin with other approaches based on gadget displacement which measures how many gadgets are displaced in the target binary after applying diversification techniques. This metric is not applicable for techniques such as [50, 52, 58, 61] that do not diversify the binary, hence the gadget displacement is zero. Marlin displaces 99.78% of the gadgets in `.text` section by using function level randomization. Thus, we show that the coarse granularity randomization is sufficient and finer granularity randomization such as instruction level or basic block level randomization does not offer any additional benefits and in some cases may lead to unnecessary overheads.

5.5 Discussion

Our proposed solution to defend against code-reuse attacks was to increase the entropy by randomizing the function blocks. One may apply this randomization technique at various levels of granularity - function level, block level or gadget level. The level of granularity to choose is a trade off between security and performance. In our implementation, we implemented the randomization at the function level which is the most coarse granularity amongst the three mentioned above. However, we show that even this coarse level of granularity provides substantial randomization to make brute force attacks infeasible.

Our prototype implementation requires the binary disassembly to contain symbol information, i.e. a non-stripped binary. In practice however, binaries may be stripped and not contain the symbol information. We address this by using external tools such as *Unstrip* [95] that restore symbol information to a stripped binary. Another

approach to process stripped binaries is to randomize at the level of basic blocks since they don't require function symbols to be identified. However, randomizing at basic block granularity will likely incur higher runtime overhead as it would break the principle of locality.

One limitation of Marlin is that it is unable to correctly rewrite certain binaries if these target binaries have certain compiler optimizations enabled or if they are obfuscated. This is because Marlin requires the *.text* section in the target binary to be organized as function blocks and for these function block to be clearly identifiable using a disassembler.

5.6 Conclusion

In this work, we proposed a fine-grained randomization based approach to defend against code reuse attacks. This approach randomizes the application binary with a different randomization for *every run*. We have implemented a prototype of our approach and demonstrated that it is successful in defeating real ROP attacks crafted using automated attack tools. We have integrated this into a custom bash shell that randomizes a binary before executing it. We have also evaluated the effectiveness of our approach and showed that the brute force effort to attack Marlin is significantly high. Based on the results of our analysis and implementation, we argue that fine-grained randomization is both feasible and practical as a defense against these pernicious code-reuse based attack techniques.

6 RUNTIME DETECTION AND RESPONSE AGAINST CODE REUSE ATTACKS

While several defense techniques have been proposed to detect and/or prevent ROP attacks, they fail to provide a complete defense system that also diagnoses and responds to these attacks in real time. Attack diagnosis is a critical part of a defense framework as it provides useful information for deploying response and preventive measures. We propose ROPShield, a defense framework that integrates detection, diagnosis and response against ROP attacks. The detection component uses a runtime monitoring mechanism to continuously verify execution constraints and raise an alarm if these constraints are violated. Based on the execution constraints that are violated, ROPShield diagnoses the type of attack and generates a report containing fine-grained diagnosis information. This diagnosis report is then leveraged by the response component to deploy appropriate response action such as generating a patch for buffer overflow. We evaluated ROPShield prototype using nine code reuse exploits collected from various sources and found it to be effective in defending against these exploits.

While attack detection or prevention techniques are effective against ROP attacks, they are not without their limitations. Firstly, they provide little or no diagnostic information about the attack. In absence of diagnostic information, very limited response actions, such as, shutdown processes, can be executed. Second, in the event of an attack, most of these techniques, such as [4, 42, 43, 96], either crash or halt the current process. The crash reports then need to be analyzed by system administrators and responses are applied manually. This may introduce significant delays in deployment of responses and the integrity of the system might have already been compromised by that time. Also, human errors such as oversight of some data may reduce credibility of the diagnosis.

A strong defense framework against ROP attacks must thus integrate components for detection, diagnosis as well as response against these attacks. The main contributions of this work are as follows.

- We propose a complete defense framework, ROPShield, for detection, diagnosis and response against ROP attacks.
- We present a novel detection technique for ROP attacks that is based on evaluating certain execution constraints instead of the complete control flow graph.
- We demonstrate how the diagnosis information generated by ROPShield can be integrated into response mechanisms.
- We present a prototype implementation of ROPShield for 32-bit x86 architecture.

This chapter is organized as follows. Section 6.1 presents a high level overview of the ROPShield framework. Section 6.2 discusses the detection and diagnosis component of ROPShield. Next, we present the details of response component in section 6.3. Section 6.4 discusses the details of the prototype implementation and presents the evaluation of our approach. Finally, we discuss limitations and advantages of our approach and conclude in section 6.5.

6.1 System Overview

Figure 6.1 shows a high level view of ROPShield and of the interactions between its components. The first component is the detection and diagnosis component that is responsible for detecting ROP attacks. It uses run-time tracing to monitor the execution of the target application and detect illegal execution. In addition to ROP attacks, ROPShield can also detect some other similar attacks such as return to libc and code injection (using stack smashing). A configuration file is used to specify the type of attacks that should be detected. By default, ROPShield will monitor and

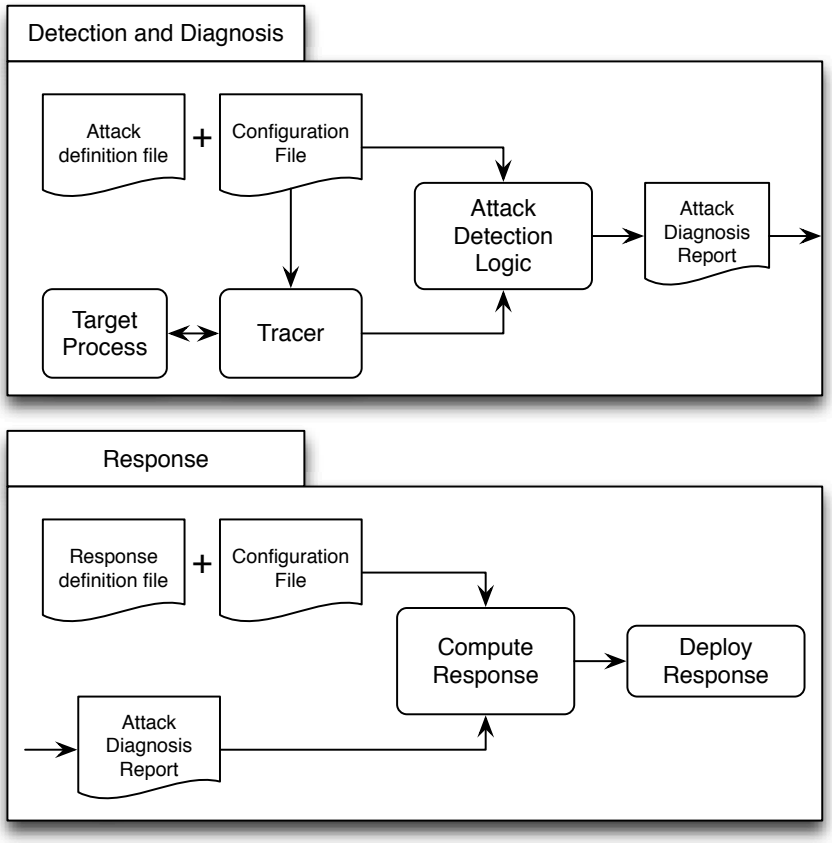


Figure 6.1. System overview

stop all supported types of attacks. ROPShield not only detects an attack, but it also identifies the type of attack based on the execution constraints that are violated. Once an attack is detected, ROPShield generates an attack diagnosis report and invokes the response mechanism. This diagnosis report contains more information about the attack such as the type of attack, and the vulnerable function, and serves as an input to the response component. The response component computes the response according to a set of predefined response rules and deploys the response actions.

```
<config>
  <attack id="CODE_INJECTION">
    <response id="KILL_PROCESS"
              mode="AUTOMATIC" >
    </response>
    <response id="ENABLE_W_XOR_X"
              mode="AUTOMATIC" >
    </response>
  </attack>

  <attack id="ROP">
    <response id="KILL PROCESS"
              mode="AUTOMATIC" >
    </response>
    <response id="PATCH_BUFFER"
              mode="AUTOMATIC" >
    </response>
    <response id="ENABLE_CODE_RANDOMIZATION
              mode="MANUAL" >
    </response>
  </attack>
</config>
```

Figure 6.2. Example of configuration file

Attack Definition File

This file specifies the attacks that are supported by ROPShield. These attacks are described in terms of various execution constraints as explained later in section 6.2.2. The current implementation supports code injection attacks, that use stack smashing, return to libc and ROP attacks. The attack definition file is write protected to prevent unauthorized modification and can only be modified by the root user. The DTD for this XML file is specified below.

```
<!ELEMENT attackdefs (attack)*>
<!ELEMENT attack (id, constraint+)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT constraint (#PCDATA)>
```

Configuration File

This file configures the operation of ROPShield according to the specific target application. This is required because different applications may have different security requirements and would need to respond differently in the event of an attack. If two or more applications have similar security requirements, then they can use the same configuration file. The configuration file to be used with the target application is specified as an input argument to ROPShield while invoking this application. If no configuration file is specified, then ROPShield uses the default configuration file.

The configuration file specifies the following information:

- The type of control flow hijacking attacks to check for a given target application. By default, all supported attacks are checked by ROPShield.
- The response to deploy for each type of attack. The default response action is to shutdown the target application when an attack is detected.
- The mode of response deployment. This can be either manual or automatic. In manual mode, the response is manually deployed by the system administrator, where as in automatic mode the response is deployed automatically without any human intervention.

The DTD for this configuration file is specified below.

```
<!ELEMENT config (attack)*>
<!ELEMENT attack (response)*>
<!ELEMENT response EMPTY>
<!ATTLIST attack
    id CDATA #REQUIRED>
<!ATTLIST response
    id CDATA #REQUIRED
    mode (AUTOMATIC|MANUAL) #REQUIRED>
```

An example of configuration file is shown in Figure 6.2. This specifies that ROP-Shield should check for code injection and ROP attacks while monitoring the target application. If a code injection attack is detected, then the response deployed according to this configuration file would be to first kill the process and then enable $W \oplus X$ protection. If ROP attack is detected, then the responses deployed automatically would be to kill the process and patch the buffer overflow. The response action `ENABLE_CODE_RANDOMIZATION` is marked as manual indicating that some interaction on behalf of the security administration, such as confirm action, may be required to deploy this response action.

Tracer

This component is responsible for monitoring the target process at run-time. The tracer is attached to the target process and monitors its execution at a very fine granularity, that is, at instruction level. It continuously feeds the trace information to the attack detection logic for verifying the execution constraints.

Attack Detection Logic

This is the algorithm that detects and identifies control hijacking by verifying execution constraints at every step of the target process' execution. The execution constraints to be evaluated are derived from the attack definition file and the attacks specified in the configuration file. The intuition and details of this detection logic are discussed in sections 6.2.1 and 6.2.2.

Diagnosis Report

If an attack is detected by the attack detection logic, then a diagnosis report is generated. This diagnosis report contains information about the type of attack detected, the vulnerable function, and other process state information at the time

of attack detection. This diagnosis information can then be used by the response component to compute and deploy the response. An example of diagnosis report generated for a ROP attack is shown later in Figure 6.4.

Response Definition File

This file contains a list of responses that are supported by ROPShield. Each response is specified as a 3-tuple that consists of a response identifier, the command to be executed as response action, and the arguments to be passed to this command. The response identifier uniquely identifies a response action and is used in the configuration file to indicate the response to be deployed for a given attack.

Response Computation Logic and Deployment

If an attack is detected, the response computation logic identifies the appropriate response identifiers according to the configuration file. Once the appropriate response has been computed, it needs to be deployed. The response actions to be invoked are identified from the response definition file using the response identifiers. If the configuration file has specified automatic deployment, then the response action command is invoked immediately. However, if the deployment mode has been marked manual, an alarm is raised and the security administrator is notified.

ROPShield is designed as an extensible framework, that is, it can be extended to include additional types of attacks and response.

6.2 Attack Detection and Diagnosis

We now discuss the attack detection logic in further detail. We start by providing the basic intuition behind our detection algorithm and then delve deeper into the details of detection logic and attack diagnosis.

6.2.1 Basic Intuition

The basic intuition behind our detection logic is that the control flow of a normal execution differs from the control flow of a subverted program where the latter violates certain execution constraints. For example, consider a ROP attack. ROP attack logic works by chaining together multiple gadgets to achieve the desired functionality. Our detection technique for ROP attacks is based on the following key observations. First, the gadgets required for an exploit payload are often scattered across different function blocks. This causes the attack execution to jump across multiple function boundaries. Second, a unique combination of instructions identify the prologue and epilogue of a function. This limits the number of gadgets that can appear in the beginning of a function which leads to our next observation. Third, most of the gadgets required in an exploit payload do not start at the entry point of a function. That is, the control flow from a compromised stack jumps to some instruction in the middle of an arbitrary function instead of the start of callee function or return address in the caller function (See Figure 6.3).

The key takeaway from these observations is that the control flow for a ROP execution differs from a normal execution in that it violates certain execution constraints. In a normal execution, when a function is called, a frame is allocated on the stack that stores the local data of the function. This stack frame represents the execution context of the function. The current instruction that is being executed in a process must belong to the function which has its frame on the top of the stack. In a ROP execution, the compromised stack contains a series of return addresses that point to instructions (or unintended instructions) in the middle of an arbitrary function. When these gadgets are executed, they do not have their own context on the stack and they are executed from a different function's stack frame rather than from their own. So, if the system starts executing instructions that do not belong to the current function context, this generally indicates that an illegal execution is taking place due to an ongoing ROP attack. There are exceptions to this rule where

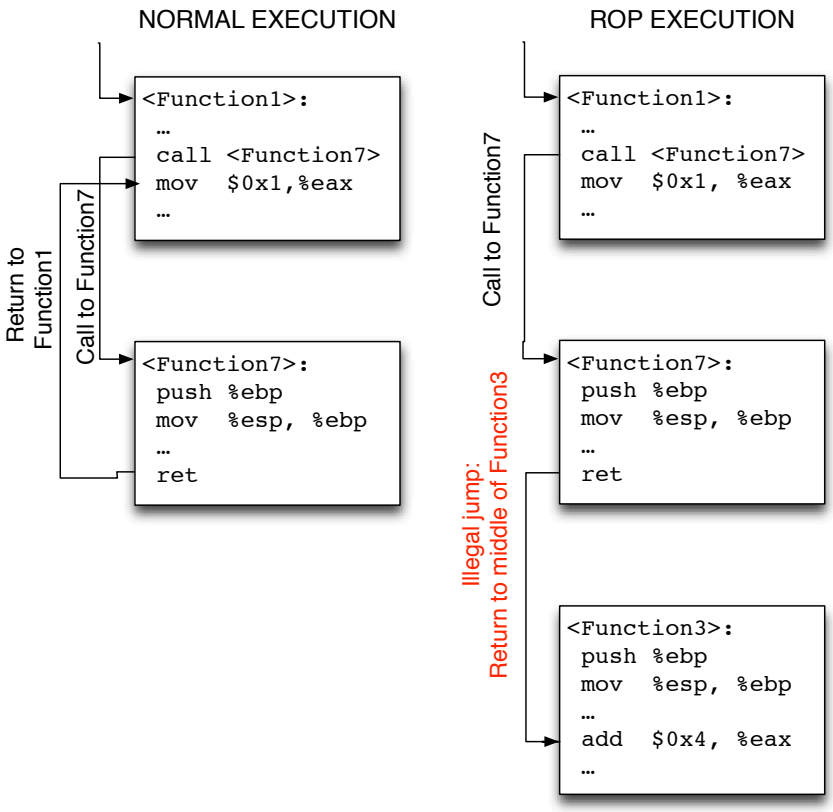


Figure 6.3. Difference in control flow between a normal execution and a ROP execution

some small functions do not establish a frame on stack. Such functions are identified during preprocessing stage and handled as a special case by our technique.

6.2.2 Detection Algorithm

We leverage the previous observations to develop a detection technique that can detect ROP attacks in real time. This technique uses a process tracing based approach to monitor and analyze the execution of the target process. The detection algorithm proceeds as follows.

Preprocessing Step

The first step is the preprocessing step that is used to identify the functions and related information in the target binary. In this step, the target binary is disassembled using a disassembler such as `objdump`. The output of the dissembler is then parsed to identify the function blocks and their bounds, that is, the start and end addresses. This information is used to associate the current instruction with its corresponding function. During the preprocessing step, we also identify those functions that do not set up a stack frame. This information is useful in detection logic to avoid incorrect detection.

Process Monitoring

The next step is tracing where the target process is traced and monitored by our detection system. The tracing functionality is achieved by using the `ptrace` call to trace the target process at instruction level granularity. We single step through each instruction and examine various variables such as current instruction pointer (`%eip`), current stack pointer (`%esp`) and the instruction being executed. We also maintain a stack of functions that correspond to the function frames currently on the stack. For each instruction that is about to be executed, we identify the function to which it belongs using the function bounds information gathered during preprocessing phase. A change in function context can be identified by checking if this function does not correspond to the function frame on the top of the stack.

Our detection logic evaluates certain execution constraints before each instruction execution to detect ROP attacks. These constraints are based on identifying illegal jumps during the process execution. The execution constraint violations that are checked by ROPShield are as follows.

C1 **Execute from Stack:** This check aims at detecting code injection attacks where the shell code is injected on the stack and the control flow is transferred

to it. Note that this constraint is relevant only for code injection attacks and not for ROP attacks which are code reuse attacks.

C2 **Return to Middle:** This checks for illegal control transfer where the callee function returns to the middle of a function that is not the caller function. This is a common behavior in ROP attacks.

C3 **Return to Start:** This checks for illegal control transfer where a function returns to the start of a function. Control transfer to beginning of a function is legal only when the preceding instruction is a jump or call to this function. This constraint is useful in detecting ROP as well as return to libc attacks.

The evaluation for constraints C2 and C3 is done only when a change of function context is detected. We assume that the jumps that occur within the boundaries of a function are legal. While this may not always be true, it is a reasonable assumption for detecting meaningful attacks since the gadgets required to craft a shell code exploit are usually not present entirely within a single function body and are typically spread across different functions.

When the next instruction to be executed belongs to a function different from the current function context on stack, we check if this is an illegal context change. A legal change of function context occurs due to one of the two reasons:

1. The callee function has finished the execution and has returned. The top of the stack will now contain the stack frame for the caller function.

2. The current function has made a function call that pushes the stack frame for the callee function on the top of the stack and change the current function context to the callee function.

In buffer overflow based control hijacking, the attacker exploits the first case by overwriting the return address to return to somewhere else instead of to the caller. This unintended control flow is detected as follows. ROPShield checks if the current instruction belongs to the middle of a function or the beginning of a function. If the

former is true and this function is not the caller function (constraint C2), then this is flagged as illegal execution. If the instruction belongs to the start of a function and the previous instruction was not a call or jump to this function (constraint C3), then this is also flagged as illegal execution. In all other cases, the function context change is passed as legal and the execution is allowed to continue to the next instruction. However, if an illegal execution is flagged, an alarm is raised and the response mechanism is deployed.

As the process tracing and detection is done in real time, our approach is effective in stopping a ROP attack before it can achieve privilege escalation. A typical shell code exploit requires multiple gadgets to construct the attack logic. All of the required gadgets are unlikely to exist in a single function and are usually spread out over multiple functions. Thus, to execute the attack logic, instructions from multiple functions must be executed from within the stack frame of the function that has the vulnerable buffer. This violates the execution constraint that is checked by our algorithm and our detection scheme is able to stop the ROP attack before it can achieve privilege escalation.

6.2.3 Attack Diagnosis

In addition to detecting the attack, ROPShield also identifies the type of attack based on the execution constraints that were violated. The current design supports three types of attacks as discussed below. ROPShield can be extended to support more types of control hijacking attacks by identifying the relevant execution constraints violated by an attack and adding this information to the attack definition file.

Attacks identified by ROPShield's diagnosis component are:

1. ROP attacks: These attacks are code reuse attacks that subvert the control flow by overwriting the return address and jump to multiple arbitrary locations

to execute desired attack logic. These attacks are identified if constraints `C2` and/or `C3` are violated.

2. Return to libc: This attack is a specific instance of ROP attack where control jumps to a function in libc after subverting the return address. This can be identified if the execution constraint `C3` is violated and the target of jump belongs to dynamically mapped libc library.

3. Code injection on stack: While this attack can be prevented by using $W \oplus X$ protection, it can still occur for systems that do not enforce this protection. These attacks can easily be detected by identifying the stack boundaries using the `/proc/[pid]/maps` file and checking if the instruction pointer `%eip` is fetching the instruction from this memory range. This corresponds to the execution constraint `C1`.

Once our framework detects and identifies the attack, it generates a diagnosis report. This report contains detailed information about the attack, namely: the type of attack detected, the input arguments, the saved state of program at time of crash (such as `%esp`, `%eip`), the vulnerable function. Tracing the target process at instruction level granularity allows ROPShield to collect good quality diagnosis information that is useful in deploying appropriate response mechanisms. Once the response actions have been deployed, the corresponding information is also appended to the diagnosis report. An example of diagnosis report generated by ROPShield is shown in Figure 6.4.

6.3 Response

There are many possible ways in which the system may respond to ROP attacks. One interesting approach is to let the illegal execution proceed and observe information that the attacker is trying to gather or the actions that the attacker is trying to execute. While this may be a good solution for systems that are set up as hon-

```
[DIAGNOSIS_REPORT] ./scpy1-bad-report.txt
[PROCESS_NAME] ./scpy1-bad
[PROCESS_PID] 21001
[ATTACK_DETECTED] ROP
[VULN_METHOD] test
[STACK_BOUNDARY] ff94c000 - ff96d000
[PREV_STATE_EIP] 8048efd
[PREV_STATE_ESP] ff96b0cc
[PREV_STATE_FUNC_NAME] test
[PREV_STATE_FUNC_START] 8048ed0
[PREV_STATE_FUNC_END] 8048efd
[CURR_STATE_EIP] 80e3042
[CURR_STATE_ESP] ff96b0d0
[CURR_STATE_FUNC_NAME] __EH_FRAME_BEGIN__
[CURR_STATE_FUNC_START] 80de4c4
[CURR_STATE_FUNC_END] 80ec8e9
[RESPONSE_DEPLOYED] KILL PROCESS
[RESPONSE_DEPLOYED] PATCH BUFFER
[SOURCE_FILE] /home/alice/Exploits/scpy1-bad.c
[VULN_FUNC_LINE_START] 34
[VULN_FUNC_LINE_END] 39
[UNSAFE_LIB_FUNC] strcpy
[UNSAFE_LIB_FUNC_LINENUM] 37
[OLD_SRC_LINE]  strcpy(buf, str);
[CORRECTED_SRC_LINE] strlcpy(buf,str,sizeof(buf));
```

Figure 6.4. Example of diagnosis report

eypots, this is clearly not a viable solution for production level and mission critical systems. This is due to the fact that allowing these attacks to proceed may lead to privilege escalation and the attacker will be able to subvert the defense and response mechanism of the system.

Different response mechanisms may be desirable for different scenarios. For example, for remote attacks against network applications such as a web server, an appropriate response would be to block the remote IP address. However, if the attacker is a local user on a system attempting privilege escalation, then blocking remote IP address will not help and the system must respond by restricting access to the application until it has been fixed, patch the vulnerability and redeploy the application. Also, the type of response deployed depends on the type of attack and the application being attacked. For example, an appropriate response for code injection attacks

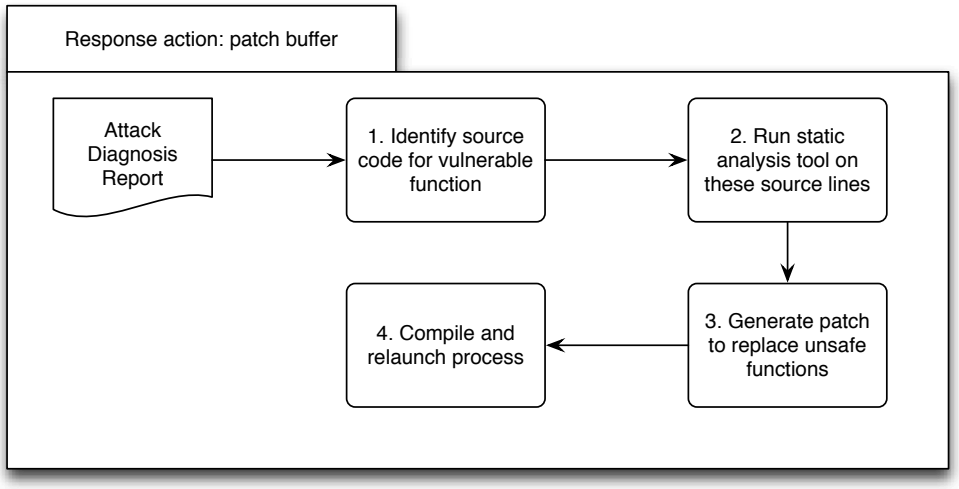


Figure 6.5. Response action: Patch buffer overflow

would be to turn on $W \oplus X$ protection, but this response will not be effective against a code reuse attack. To allow for this flexibility, ROPShield uses a configuration file that allows one to specify the response rules that determine the response actions to be deployed when a certain type of attack is detected. These response rules also indicate whether the response should be deployed manually or automatically.

6.3.1 Response Actions

When an attack is detected, a post-attack response must be deployed. As mentioned earlier, this can be either short term response or long term response. In this section, we present two long term response actions - *patch buffer overflow* and *deploy code randomization* in further detail. We do not discuss short term response actions (such as kill target process) as their implementation is quite straightforward and does not pose any interesting challenges.

6.3.2 Response Action: Patch Buffer Overflow

This long term response action aims at patching the buffer overflow vulnerability that is exploited by the ROP attack. Our approach targets unsafe library functions such as `strcpy` that copies `char` data from a source buffer to a destination buffer until it encounters a null character. `strcpy` does not do any bounds check while copying this data. This allows a malicious entity to overflow the destination buffer and overwrite return address to hijack the control flow of victim application.

This response action leverages the fine grained diagnosis information generated during the diagnosis phase to identify and patch the buffer. Figure 6.5 indicates the various steps involved in deploying this response action.

Running Example

In the discussion that follows, we use the following running example to illustrate our approach.

```
36  void vuln (char* buff) {
37      char tmp[8] = { '\0' };
38      strcpy(tmp, buff);
39      printf("-> %s\n", tmp);
40  }
```

In this example, `vuln` is a vulnerable function with buffer overflow vulnerability. It declares a `char` buffer, `tmp`, that is statically allocated with 8 bytes (line 37). This function calls the unsafe library function `strcpy` in line 38 that copies the contents of buffer `buff` to buffer `tmp`. The `strcpy` function does not perform any bounds check while copying the buffer. Thus, if `buff` is longer than 8 bytes and does not contain a null terminating character, then the execution of the function will overwrite the data on the stack. This can be used to craft malicious attack where the attacker supplies

a carefully crafted `buff` to overwrite the return address on stack and redirect control to the first gadget of the ROP exploit code.

Identifying the Vulnerability

The first step of this response action is to identify the exact vulnerability in the source code of the application. When ROPShield detects an illegal control flow due to ROP, it also identifies the function that contains the vulnerable buffer. At this stage, only the address in the ELF binary and the corresponding symbol name are known. In our running example ROPShield would produce the following diagnosis information:

```
...
[Vulnerable function] vuln()
[Start address] 0x08048ef6
[End address] 0x08048f31
...
```

This diagnosis information indicates that ROPShield detected a vulnerability in the `vuln` function between addresses 0x08048ef6 and 0x08048f31. These addresses are the function boundaries.

This information along with the debugging information in the ELF binary is used to identify the corresponding line numbers in the source file. Specifically, the response action invokes the `addr2line` tool to extract this information from the ELF binary. `addr2line` tool translates the addresses into source file names and line numbers. For the above example, this tool would produce the following output:

```
0x08048ef6: vuln at /home/alice/test/example.c:37
0x08048f31: vuln at /home/alice/test/example.c:42
```

This means that the vulnerable function `vuln` is contained between lines 37 and 42 in the source file example.c.

Once this information is obtained, the response action analyzes these line numbers by invoking a static analysis tool which identifies unsafe library functions and static buffers that are used in this line number range. The advantage of using analysis tools on these line numbers rather than the entire source is that this will be more accurate and efficient. The modular design of ROPShield allows integration of various static analysis tools to perform this step. For our implementation, we chose to integrate `flawfinder` tool [99] in this static analysis step.

`Flawfinder` analyzes the source code according to an existing set of rules to find the potential vulnerability. Using this tool on the source files and line numbers identified in the above step gives the following output:

```
test/ndh_rop.c:40:  [4] (buffer) strcpy: Does not check for buffer
overflows when copying to destination. Consider using strncpy or
strlcpy (warning, strncpy is easily misused).
test/ndh_rop.c:38:  [2] (buffer) char: Statically-sized arrays can be
overflowed. Perform bounds checking, use functions that limit length,
or ensure that the size is larger than the maximum possible length.
```

This information indicates that the unsafe library function `strcpy` is used in line 38 and can lead to potential buffer overflow. Further, it also identifies the buffer `tmp` in line 37 as a statically allocated buffer that can be overflown. Once these unsafe functions have been identified, the response action proceeds to the next step, that is, generating the source patch as discussed below.

Generating the Source Patch

The next step after the unsafe function is identified is to patch the source code with the safe variant of this function. The safe variant of this function takes the length of the destination buffer as input to perform bounds checking. Before replacing the unsafe function with its safe variant, the response action must estimate the destination buffer size. If the destination buffer is a statically allocated `char` buffer, then the

buffer size is computed using the `sizeof(.)` function on the destination buffer. However, if the destination buffer is a char pointer, then one needs to perform a backward analysis to identify the statically allocated char buffer that it points to before applying the `sizeof` operator. Following is an example of this scenario.

```
400 void func (char *src) {
401     char dest[25] = { '\0' };
...
410     char *destptr = dest;
...
417     strcpy (destptr, src);
418 }
```

The response action now replaces the unsafe libc function with its safe variant. In our example above, `strcpy` is replaced with `strlcpy` that performs bounds checking while copying to the char buffer. The function call in line 38 of our working example is now replaced with:

```
38    strlcpy(tmp, buff, sizeof(tmp));
```

After this patch is applied to the source code, the source code is recompiled and the application is restarted. Since all the above steps are performed automatically by ROPShield's response component, the availability of the victim application is significantly improved when compared to the traditional approach where a security administrator has to manually examine log files and application code to identify the vulnerability and fix it.

6.3.3 Response Action: Deploy Code Randomization

Another long term response action is to deploy preventive security mechanisms that will strengthen a system against ROP attacks. One such mechanism is code randomization which defends against ROP attacks even in the presence of buffer overflow vulnerability.

The underlying assumption behind any ROP attack is that the attacker is aware of the entire layout of the process. That is, he/she knows the exact address of each instruction in the executable code. This is used to identify useful gadgets and craft the exploit payload. A code randomization-based ROP defense technique called Marlin [96] can be used to break this assumption and thwart ROP attacks. Marlin randomizes an application's code by shuffling around the code blocks according to a random permutation. This randomization is performed at run-time, that is, the application executes with a different code layout with every execution. This denies attacker any knowledge of the addresses where various instructions are loaded, thus making it impossible to craft a valid ROP exploit.

Marlin uses a whitelist to specify which applications to randomize with every execution. This whitelist serves as the integration point for using Marlin as a response action within ROPShield. When ROPShield detects an ROP attack against an application, the application is terminated, added to Marlin's whitelist and then restarted. This new execution of the vulnerable application uses a randomized layout, thus thwarting future ROP attempts.

Integrating the Marlin randomization approach with ROPShield has multiple advantages. First, it enables a selective application of the Marlin technique on a demonstrated need basis. Security mechanisms are often considered prohibitive which leads to these mechanisms to not be used in the first place. However, applying such defense mechanisms in response to detected attacks can improve the adoption of such techniques. Second, ROPShield can be deployed on both randomized (according to Marlin) and unrandomized binaries. While running ROPShield on a Marlin-processed binary provides an extra degree of security, using only Marlin without ROPShield can provide better efficiency. Depending on the application requirements, the administrator can choose to turn off ROPShield once the randomization based response using Marlin has been deployed.

6.4 Implementation and Evaluation

We have implemented a prototype of ROPShield for 32-bit x86 architecture on Ubuntu 12.04 operating system. We now discuss the implementation details of ROPShield.

6.4.1 Detection and Diagnosis Component

The first step of the ROPShield detection component is the preprocessing step that is responsible for identifying the function block boundaries in the executable code. We use the `objdump` utility to disassemble the binary and parse symbol information along with the start and end address of each symbol. Another information retrieved during this stage is the identification of functions that execute without establishing their own stack frame. We noticed that some small functions in libc such as `__libc_read` demonstrated this behavior. Since these functions do not establish their own stack frame, they execute from the function context of the caller function. Since ROPShield identifies this as illegal execution, this resulted in false positives. To address this issue, we identify and mark such functions during preprocessing stage to exempt these functions from the constraint evaluation. This identification is executed by searching for instruction sequence `push %ebp; mov %ebp, %esp` which is the standard instruction sequence to establish a stack frame for a function. This preprocessing step needs to be executed just once per application and can be performed offline. The information generated during this phase can be used for future runs of the application.

To protect an application using ROPShield, we need the ability to continuously monitor its execution. This is implemented by invoking the target application using a wrapper that allows ROPShield to trace it, that is, the target application is forked as a child process of ROPShield and tracing is executed by invoking the `PTRACE_TRACEME` request. For the tracing functionality, we decided to write our own tracer instead of using the available trace utilities or debuggers such as `ltrace`, `strace`, `gdb`. The

reason is that we need to control and monitor the execution at a much finer granularity without incurring significant overhead. Our implementation of tracer uses the `ptrace` system call that is the foundation of several tracing and debugging tools including `ltrace`, `strace` and `gdb`. We introduce a breakpoint just before the `main()` function call which is the point after which the process execution needs to be carefully monitored. Once the `main()` function is encountered, we single step through the instructions using the `PTRACE_SINGLESTEP` mode of the `ptrace` method.

At each instruction execution, we monitor the following variables: the stack pointer (`%esp`), the instruction pointer (`%eip`), and the current instruction being executed. The current instruction being executed is retrieved from `/proc/[pid]/mem` file for the target process. We also maintain a stack of functions that corresponds to the function frames on the real stack. This is used for verifying the execution constraints `C2` and `C3`.

In our initial implementation we used `distorm3` [100], a disassembler library, to disassemble and decode the instructions retrieved in each step. This can lead to unnecessary overhead as we are only interested in knowing the instruction type of an instruction to identify whether it is a control flow instruction (such as jump, call or ret). To get this information from `distorm3`, we had to disassemble the complete instruction. To eliminate this overhead, we wrote our own lightweight disassembler that only looks at instruction opcodes to determine the type of instruction without disassembling it completely.

While implementing the prototype for ROPShield, we had to handle some special cases that are exceptions to typical control flow or use uncommon coding conventions. Functions that execute without a stack frame is one example of this. Another exception is a special return sequence, `repz ret` that is used instead of just `ret` for performance reasons.

ROPShield uses various files such as configuration file and attack definition file to configure its behavior. If an attacker can modify these files, he/she can easily bypass ROPShield protection. Thus it is important that these files be stored securely. In

our current implementation, these files are stored in the installation directory of ROPShield and are write-protected so that they can be modified only by root user.

6.4.2 Response Component

ROPShield integrates various response actions that are specified in the response definition file. The structure of response definition file (see section 6.1) makes it trivial to add new response actions which can be implemented as separate modules and then invoked from ROPShield.

The current implementation of ROPShield supports various response actions such as process termination, process restart, patch buffer overflow, and code randomization. Each response is implemented as a separate perl script that extracts the required information from the diagnosis report to process its response logic. The command to invoke this perl script along with its arguments is specified in the response definition file. This response definition file is also write-protected and is writable only by the root user. When ROPShield detects an attack, it reads the configuration file to identify the responses to be deployed, their mode of deployment (automatic or manual) and the order according to which responses must be deployed. If a response action in this response sequence is marked with manual deployment, ROPShield raises an alarm for security administrator and continues deploying the remaining automatic responses.

One of the response actions, patch buffer overflow, is a long term response that aims at permanently fixing the vulnerability in the source code. Figure 6.5 provides an overview of this response action. The first step, that is, the identification of the source file requires debug headers to be present in the ELF executable. This requires that the source code is compiled using the debug flag (`-g`). We use the `addr2line` utility to get the source file and line number mapping for a given instruction address. We provide the start and end address of the function symbol (as obtained during preprocessing stage, see section 6.4.1) as input to `addr2line` to get the line numbers

corresponding to the source of vulnerable function. Once we have this information, we use `flawfinder`, a static analysis tool, on this source file and extract only those hits that correspond to the lines in this function's source. This helps in eliminating several false positives that are detected by `flawfinder`. Next, we analyze the hit list to identify the use of unsafe library functions and statically allocated buffers. These functions are then replaced with their safe variants that use the estimated destination buffer size to prevent buffer overflow. All these steps are integrated into one perl script that is invoked by ROPShield while deploying this response action.

The substitution of unsafe library functions with their safe variants is done according to a set of replacement rules. The functions `strcpy`, `strcat` and `sprintf` are replaced with `strlcpy`, `strlcat` and `snprintf`, respectively, that take an additional argument which is the destination buffer size. The vulnerable function `gets` is replaced with `fgets` that takes two additional arguments – the destination buffer size and input stream (in this case, `stdin`). Further, for functions such as `strncpy` or `snprintf` that already take a size argument as input, the size argument may be incorrectly specified by the developer. If such a function is detected, then the size parameter is replaced with a ternary expression that takes the minimum of original size argument and the estimated size of destination buffer. Also, for cases where the substituted function is either `strlcpy` or `strlcat`, the patched program needs to be compiled with `libbsd` since these functions are not a part of standard libc on Linux systems.

Our design leverages existing analysis tools such as `flawfinder` to identify vulnerable function. Thus, the accuracy of our diagnosis and response components depends from the accuracy of the external tools that we integrate. However, the fine grained diagnosis information generated by ROPShield reduces the scope of search, thus increasing the accuracy of the tool. The `flawfinder` tool can be replaced by other analysis tools as well that may provide more precise information. One advantage in using `flawfinder` is that all vulnerable functions identified in our search scope will

be fixed, rather than just fixing one vulnerable function that was responsible for the overflow.

To fix the buffer overflow automatically, an obvious requirement is that the source code be available on the system on which ROPShield is executing. However, in some scenarios (for example, proprietary code) it may not be possible to make the source code available on the deployed system. The response action for fixing buffer overflow, as discussed above, only takes the diagnosis report as input. Thus, in such scenarios, one can consider an architecture where the response component will send this diagnosis report to an application maintenance server as feedback. The application maintenance server, that contains the source code will fix the buffer overflow vulnerability in source, recompile the software and then push the updated application over the network. That is, when the source code is unavailable, the patch can be generated remotely and deployed to multiple systems at once.

6.4.3 Evaluation

We evaluated the effectiveness of detection, diagnosis and response components of ROPShield by testing it with various exploits (see section 6.4.3). The tests were performed on a 32-bit x86 machine with Ubuntu 12.04 operating system.

Exploit dataset

To evaluate the effectiveness of our approach, we tested ROPShield against 9 code-reuse exploits that were collected from different sources. These exploits are summarized in Table 6.1. The first two exploits, RG1 and RG2, are from the test cases that were included with the ROPgadget attack tool v4 [98]. ROPgadget is a ROP attack tool able to automatically find relevant gadgets and create the attack payload. The first attack payload (referred to as 'shellcode 1' in Table 6.1) spawned a shell while the second attack made a bind shell code to listen at a specified port.

Table 6.1.
Exploit summary

ID	Exploit Source	Vulnerability cause	Exploit Type	Payload
RG1	ROPgadget v4 testcase	Unsafe function `strcpy`	ROP	shellcode 1
RG2	ROPgadget v4 testcase	Unsafe function `strcpy`	ROP	bind port
EDB1	Exploit-DB#17286(a) [101]	Unsafe function `strcpy`	Ret-to-libc	shellcode 2
EDB2	Exploit-DB#17286(b) [101]	Unsafe function `strcpy`	Ret-to-libc	shellcode 3
SRD1	NIST SRD Testcase #1563	Unsafe function `gets`	ROP	shell code 1
SRD2	NIST SRD Testcase #1600	Unsafe function `strcpy`	ROP	shellcode 1
SRD3	NIST SRD Testcase #1616	Incorrect size in `snprintf`	ROP	shellcode 1
SRD4	NIST SRD Testcase #1636	Unsafe function `sprintf`	ROP	shellcode 1
SRD5	NIST SRD Testcase #2081	Unsafe function `strcat`	ROP	shellcode 1

Both these attacks bypass ASLR protection as the gadgets were constructed from application's code and not library code.

The next two attacks, EDB1 and EDB2, are from Exploit Database (EDB) [102] which is a repository for exploits and vulnerable software. For the EDB exploits, we followed the instructions in the documentation to manually create our own vulnerable application and exploit. These EDB exploits that we used are advanced return to libc attacks that bypass ASCII armor protection. The two variants of this attack use different shell code. The first one (referred to as 'shellcode 2' in Table 6.1) uses a hardcoded address of "/bin/sh" string from shell environment. The second variant (referred to as 'shellcode 3' in Table 6.1) is more robust as it uses calls to `strcpy` to dynamically create the "/bin/sh" string by copying each ASCII character from a different location.

The next five attacks, SRD1-SRD5, were constructed for vulnerable code samples taken from the NIST SAMATE's Reference dataset [103]. We selected five test cases with weakness CWE-121, that is, Stack based buffer overflow. Not all code samples with this vulnerability are usable for our evaluation since in some cases the buffer overflow can only be used to crash a program and not to hijack the control flow. Examples of such test cases include incorrect length checking such as off-by-one error.

We only selected the test cases that could be exploited to launch a meaningful ROP attack. Also, as can be seen in column 3 of Table 6.1, we selected test cases with different vulnerable functions. For each of these test cases, we used the ROPgadget v4 attack tool to craft the exploit payload ('shellcode 1') to spawn a shell.

We evaluated ROPShield on these exploits and the results are summarized in Table 6.2

Exploit Detection

For each of these test cases, we executed the vulnerable program with ROPShield tracing enabled and provided exploit payload as input. As shown in Table 6.1, our test cases used 4 different type of exploit payloads amongst 9 test cases to test ROPShield with different exploit payloads. ROPShield was successful in detecting and stopping all of the nine attacks. All of these attacks violated the execution constraints by returning to another function that was not in the call stack. We also tested ROPShield with some non-malicious inputs and did not find any false positives.

Exploit Diagnosis

Once ROPShield detected these exploits, it also generated a diagnosis report for each exploit. The diagnosis report generated by one of these test cases, SRD2 is shown as an example in Figure 6.4. As shown in column 3 of Table 6.2, ROPShield correctly diagnosed the type of attack that was stopped. In addition, it also correctly identified other diagnosis information such as the vulnerable function. We also tested these samples with random long strings as input that overflow the buffer and crash the program but do not have any attack logic encoded in them. In these cases, ROPShield halts the execution of target application and does not label this as an attack. Hence, this information can be used to differentiate between an attack payload and a random crash due buffer overflow.

Response Evaluation

The response component of ROPShield uses the diagnosis report generated in previous step as input to deploy appropriate response. We evaluated the correctness of response component that is responsible for fixing buffer overflow. In all of the nine test cases discussed above, ROPShield was able to correctly identify the source code line responsible for overflow and correctly patch it. Most of these patches involved replacing an unsafe library function with its safe variant that performs bounds checking. A summary of these substitutions in presented in last column of Table 6.2. One interesting case was SRD3 that was using `snprintf` with a size parameter, but the size argument passed to this function call was incorrect. Following is a code snippet from this test case where line 37 allows at most 1024 bytes to be copied to buffer `buf` that has a maximum capacity of 40 bytes.

```
31 #define MAXSIZE 40
33 void test(char *str)
34 {
35 char buf[MAXSIZE];
...
37 snprintf(buf,1024, "%s", str);
...
39 }
```

In the above test case, the response action replaced line 37 with the corrected buffer size as shown below:

```
37 snprintf(buf,((sizeof(buf)<1024)?sizeof(buf):1024), "%s", str);
```

After the response action fixed the vulnerabilities in these 9 test cases, we restarted these test cases and tested them again with the exploit code. All the patched programs truncated the exploit code to the size of buffer and prevented buffer overflow. Thus, the response action was able to correctly identify and fix the buffer overflow vulnerabilities in these test cases.

Table 6.2.
ROPShield evaluation

ID	Detected by ROPShield	Diagnosis of Exploit Type	Identification of Unsafe Function	Response deployed: Fix buffer overflow
RG1	Yes	ROP	`strcpy`	Replace `strcpy` with `strlcpy`
RG2	Yes	ROP	`strcpy`	Replace `strcpy` with `strlcpy`
EDB1	Yes	Ret-to-libc	`strcpy`	Replace `strcpy` with `strlcpy`
EDB2	Yes	Ret-to-libc	`strcpy`	Replace `strcpy` with `strlcpy`
SRD1	Yes	ROP	`gets`	Replace `gets` with `fgets`
SRD2	Yes	ROP	`strcpy`	Replace `strcpy` with `strlcpy`
SRD3	Yes	ROP	`snprintf`	Correct size parameter for `snprintf`
SRD4	Yes	ROP	`sprintf`	Replace `sprintf` with `snprintf`
SRD5	Yes	ROP	`strcat`	Replace `strcat` with `strlcat`

6.5 Conclusion

In this work we have proposed a comprehensive framework for detecting, diagnosing and responding against ROP attacks. The detection component uses a novel technique for identifying ROP execution based on evaluating certain execution constraints. The diagnosis component is responsible for identifying the type of attack and collecting diagnostic information to pass on to the response component. We have implemented a prototype of ROPShield framework and present an evaluation of our approach using this prototype implementation.

Our detection approach offers several advantages over existing defense techniques. First, the detection component operates directly on the executable and does not require the source code of the target application. The source code is only required when the response action needs to generate a patch for source code. Second, unlike other control flow analysis approaches, our technique does not require the complete control flow graph (CFG) of the application as we check for certain execution constraints instead of verifying the control flow integrity against the complete CFG. Third, our technique does not require any code instrumentation or modification in the execution semantics. Thus, the target process executes normally and our technique observes it

without modifying its execution path. Fourth, our technique detects and stops ROP attacks in real time. This is very important as it makes it possible to diagnose the cause of the attack and respond to it in a suitable manner. Diversification based techniques that randomize the target binary will crash the application when ROP attack fails. These diversification based approaches do not provide enough diagnostic information and the diagnosis cannot be done in real time.

While our approach promises several advantages, it also has certain limitations. Correct identification of function blocks is an important requirement for our detection technique. Thus, our detection technique may not work with obfuscated code and stripped binaries where it is not possible to distinguish the function boundaries. Also, our technique requires the code for target binary to be modular. If the vulnerable buffer is present in a very large function, then the attacker might be able to build an exploit payload using addresses within that function itself without the need to switch function contexts during attack execution. Lastly, since the implementation of ROPShield's detection component uses tracing technique based on `ptrace`, the traced application will incur a performance hit during runtime. However, this overhead can be reduced by using alternate tracing techniques [104, 105] that are more efficient.

7 SUMMARY

In this dissertation, we have discussed various aspects of enforcing contextual access control in a secure manner. First, we proposed a context profiling framework to automatically infer contexts of interest for a user and intuitively configure access control policies. Since presence of other users in proximity is an important contextual factor for access control applications, we proposed a formal proximity model for RBAC systems. We applied this proximity model to realms other than just geographical proximity such as social proximity and cyber proximity.

For securing the policy enforcement system against code reuse attacks, we proposed a fine grained randomization based technique that diversifies a binary with every execution. Our proposed solution is applicable for various systems based on x86 architecture, but can be extended to other architecture as well, for instance ARM and AVR architectures. Embedded systems such as those based on Harvard architecture have been shown to be vulnerable to code reuse attacks [90]. Extending our approach to secure these systems against code reuse attacks will involve architecture specific implementation and tackling unique challenges posed by these systems such as adapting our approach for constrained resources.

We also proposed ROPShield defense framework that uses run-time monitoring to detect a ROP attack in real time and generates an attack diagnosis report to identify the type and cause of the attack. It uses this diagnosis report to deploy appropriate response such as patching the buffer overflow vulnerability. Runtime attack diagnosis is a novel approach with several interesting research directions. For instance, diagnostic information can be used to perform attack forensics, that is, understand the purpose of an attack. This type of attack forensics allows deploying appropriate responses to protect the sensitive data that is the target of the attack. Attack diagnosis can also be used to apply preventive responses to applications that

have not yet been attacked. For example, response actions can automatically be applied to applications that are similar to the application that was attacked. This would require understanding how to identify similar applications. Our current work focuses on buffer overflow attacks but future research directions can include extending this attack diagnosis to other types of attacks as well, such as heap overflow and integer overflow attacks.

REFERENCES

REFERENCES

[1] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, January 2001.

[2] Michael S. Kirkpatrick, Maria Luisa Damiani, and Elisa Bertino. Prox-RBAC: A proximity-based spatially aware RBAC. In *Proceedings of 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, pages 339–348, 2011.

[3] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *Proceedings of 3rd International Symposium on Advances in Spatial Databases (SSD)*, pages 277–295, London, UK, 1993. Springer-Verlag.

[4] 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.

[5] Long Vu, Quang Do, and Klara Nahrstedt. Jyotish: A novel framework for constructing predictive model of people movement from joint wifi/bluetooth trace. In *9th IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2011.

[6] Changqing Zhou, Dan Frankowski, Pamela Ludford, Shashi Shekhar, and Loren Terveen. Discovering personally meaningful places: An interactive clustering approach. *ACM Transactions on Information Systems (TOIS)*, 25, July 2007.

[7] Petteri Nurmi and Sourav Bhattacharya. Identifying meaningful places: The non-parametric way. In *Proceedings of the 6th International Conference on Pervasive Computing*, Pervasive '08, pages 111–127. Springer-Verlag, 2008.

[8] Eric Paulos and Elizabeth Goodman. The familiar stranger: Anxiety, comfort and play in public places. In *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 223–230. ACM, 2004.

[9] Rachel Greenstadt and Jacob Beal. Cognitive security for personal devices. In *Proceedings of the 1st ACM Workshop on AISec*, pages 27–30. ACM, October 2008.

[10] Markus Jakobsson, Elaine Shi, Philippe Golle, and Richard Chow. Implicit authentication for mobile devices. In *Proceedings of the 4th USENIX Conference on Hot Topics in Security*, HotSec'09, Berkeley, CA, USA, 2009. USENIX Association.

[11] George Danezis. Inferring privacy policies for social networking services. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, AISec '09, pages 5–10, New York, NY, USA, 2009. ACM.

[12] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for Android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 331–345. Springer-Verlag, 2011.

[13] Patrick Gage Kelley, Paul Hankes Drielsma, Norman Sadeh, and Lorrie Faith Cranor. User-controllable learning of security and privacy policies. In *Proceedings of the 1st ACM Workshop on AISec*, AISec '08, pages 11–18, New York, NY, USA, 2008. ACM.

[14] W. Keith Edwards, Erika Shehan Poole, and Jennifer Stoll. Security automation considered harmful? In *NSPW '07: Proceedings of the 2007 Workshop on New Security Paradigms*, pages 33–42. ACM, 2008.

[15] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[16] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 10(1), February 2007.

[17] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A location-aware role-based access control model. In *Proceedings of International Conference on Information Systems Security (ICISS)*, pages 147–161, 2006.

[18] Avigdor Gal and Vijayalakshmi Atluri. An authorization model for temporal data. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 144–153, New York, NY, USA, 2000. ACM.

[19] Subhendu Aich, Shamik Sural, and Arun K. Majumdar. STARBAC: Spatiotemporal role based access control. In *OTM Conferences*, 2007.

[20] S. Chandran and J. Joshi. LoT RBAC: A location and time-based RBAC model. In *Proceedings of 6th International Conference on Web Information Systems Engineering (WISE)*, pages 361–375. Springer-Verlag, 2005.

[21] Vijayalakshmi Atluri and Soon Ae Chun. A geotemporal role-based authorisation system. In *International Journal of Information and Computer Security*, volume 1, pages 143–168, 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 6th ACM Symposium on Access Control Models and Technologies (SACMAT '01)*, pages 10–20, 2001.

[23] 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.

[24] Guangsen Zhang and Manish Parashar. Context-aware dynamic access control for pervasive applications. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 21–30, 2004.

[25] Frode Hansen and Vladimir Oleschuk. SRBAC: A spatial role-based access control model for mobile systems. In *Proceedings of 8th Nordic Workshop on Secure IT Systems (NORDSEC)*, pages 129–141, October 2003.

[26] Michael S. Kirkpatrick and Elisa Bertino. Enforcing spatial constraints for mobile RBAC systems. In *Proceedings of 15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 99–108, New York, NY, USA, 2010. ACM.

[27] Christian S. Jensen, Hua Lu, and Bin Yang. Indoor: A new data management frontier. *IEEE Data Engineering Bulletin*, 33(2):12–17, June 2010.

[28] Christian S. Jensen, Hua Lu, and Bin Yang. Graph model based indoor tracking. In *10th International Conference on Mobile Data Management (MDM)*, pages 122–131, 2009.

[29] Christos K. Georgiadis, Ioannis Mavridis, George Pangalos, and Roshan K. Thomas. Flexible team-based access control using contexts. In *Proceedings of 6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 21–27, New York, NY, USA, 2001. ACM.

[30] S. M. Didar-Al-Alam, Hasan Mahmud, and Md. Abdul. Mottalib. Modifications in proximity based access control for multiple user support. *International Journal of Engineering Science and Technology*, 2:3603–3613, 2010.

[31] Sandeep K. S. Gupta, Tridib Mukherjee, Krishna K. Venkatasubramanian, and T. B. Taylor. Proximity based access control in smart-emergency departments. In *Proceedings of 4th Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW)*, pages 512–, Washington, DC, USA, 2006. IEEE Computer Society.

[32] 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, 1998.

[33] Vendicator. StackShield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

[34] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers*, 55(10):1271–1285, October 2006.

[35] Timothy K. Tsai and Navjot Singh. Libsafe: Protecting critical elements of stacks. 2001.

[36] Sandeep 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, 2003.

[37] PaX Team. PaX. `http://pax.grsecurity.net/`.

[38] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th Conference on USENIX Security Symposium – Volume 14*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[39] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society.

[40] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society.

[41] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, pages 601–615, 2012.

[42] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[43] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[44] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium – Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[46] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)*, 28:4:1–4:54, July 2010.

[47] MSDN Microsoft. /ORDER (Put Functions in Order). `http://msdn.microsoft.com/en-us/library/00kh39zz.aspx`.

[48] MSDN Microsoft. Profile-guided Optimizations. `http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx`.

[49] 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.

[50] 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.

[51] 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.

[52] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[53] 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.

[54] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[55] Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[56] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40, November 2009.

[57] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *ACM Conference on Computer and Communications Security (CCS)*, pages 308–317. ACM Press, 2004.

[58] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[59] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 311–322, New York, NY, USA, 2013. ACM.

[60] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[61] 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.

[62] Valgrind Team. Valgrind. `http://www.valgrind.org/`.

[63] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[64] Alexey Smirnov and Tzi-cker Chiueh. DIRA: Automatic detection, identification and repair of control-hijacking attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[65] Alexey Smirnov and Tzi-cker Chiueh. Automatic patch generation for buffer overflow attacks. In *Proceedings of the Third International Symposium on Information Assurance and Security*, IAS '07, pages 165–170, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, November 2005.

[67] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. AutoPaG: Towards automated software patch generation with source code root cause identification and repair. In *ASIACCS '07: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, pages 329–340, New York, NY, USA, 2007. ACM.

[68] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[69] Markus Miettinen and N. Asokan. Towards security policy decisions based on context profiling. In *Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security*, AISec '10, pages 19–23. ACM, 2010.

[70] Abigail Barr. Familiarity and trust: An experimental investigation. CSAE Working Paper Series 1999-23, Centre for the Study of African Economies, University of Oxford, 1999.

[71] Jie Zhang. Familiarity and trust: Measuring familiarity with a web site. In *Proceedings of the 2nd Annual Conference on Privacy, Trust and Security (PST 2004*, pages 23–28, 2004.

[72] Patsy A. Klaus and Cathy T. Maston. Criminal victimization in the united states, 2006, statistical tables. *National Crime Victimization Survey*, 2008.

[73] Niko Kiukkonen, Jan Blom, Olivier Dousse, and Juha.K Laurila. Towards rich mobile phone datasets: Lausanne data collection campaign. In *ICPS 2010: The 7th International Conference on Pervasive Services*, 2010.

[74] Nokia Research Center Lausanne. Lausanne data collection campaign. `http://research.nokia.com/page/11367`, 2011.

[75] Robert Goodell Brown. *Smoothing, Forecasting and Prediction of Discrete Time Series*. Dover Phoenix Edition, 2004.

[76] Victoria Bellotti and Keith Edwards. Intelligibility and accountability: Human considerations in context-aware systems. *Human-Computer Interaction*, 16:193–212, 2001.

[77] 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.

[78] Oasis. OASIS extensible access control markup language (XACML). *Spring*, 2009(May 5):1–16, 2004.

[79] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of 3rd Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[80] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32:374–382, April 1985.

[81] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 2, 1983.

[82] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of 11th Annual International Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140, London, UK, 1992. Springer-Verlag.

[83] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996.

[84] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, August 1997.

[85] Nergal. The advanced return-into-lib(c) exploits (pax case study). *Phrack Magazine, 58(4)*, December 2001.

[86] 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.

[87] 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.

[88] 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.

[89] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

[90] 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.

[91] 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.

[92] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[93] 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.

[94] 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.

[95] Paradyn Project. UNSTRIP. `http://paradyn.org/html/tools/unstrip.html`, 2011.

[96] Aditi Gupta, Sam Kerr, Michael S. Kirkpatrick, and Elisa Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. In *The 7th International Conference on Network and System Security (NSS 2013)*, June 2013.

[97] Unixbench. Byte-unixbench: A unix benchmark suite. `https://code.google.com/p/byte-unixbench/`.

[98] Jonathan Salwan. ROPgadget Tool. `http://shell-storm.org/project/ROPgadget/`, 2011.

[99] David Wheeler. Flawfinder website. `http://www.dwheeler.com/flawfinder/`.

[100] Gil Dabah. Distorm3: Powerful disassembler library for x86/AMD64. http://code.google.com/p/distorm/, 2012.

[101] Sickness. Linux exploit development part 4 – ASCII armor bypass + return-to-plt. http://www.exploit-db.com/wp-content/themes/exploit/docs/17286.pdf.

[102] Offensive Security. The Exploit Database (EDB). http://www.exploit-db.com/.

[103] NIST. NIST SAMATE reference dataset project. http://samate.nist.gov/SRD/.

[104] ERESI. The embedded ELF tracer : Etrace. http://www.eresi-project.org/wiki/TheEmbeddedELFtracer.

[105] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

VITA

VITA

Aditi Gupta was born and raised in Ranchi, India. She completed her PhD from Purdue University in 2014, where her major advisor was Prof. Elisa Bertino. She received her bachelor's and master's degrees from Indian Institute of Technology, Kanpur in 2008 where she majored in computer science and engineering. Her primary research interests include system security, context aware applications and mobile security. During her stay at Purdue, she pursued internships with Nokia Research Center in Helsinki (June-December 2010, May-August 2011) and Amazon (May-August 2012).