

Improving Trust in Cloud, Enterprise, and Mobile Computing Platforms

Nuno Miguel Carvalho Santos

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht Saarbrücken, November 28, 2013

Day of Colloquium: 27/11/2013
Dean of the Faculty: Univ.-Prof. Dr. Mark Groves

Chair of the Committee: Prof. Dr. Michael Backes
First Reviewer: Prof. Dr. Rodrigo Rodrigues
Second Reviewer: Prof. Dr. Peter Druschel
Third Reviewer: Dr. Himanshu Raj
Academic Assistant: Dr. Krishna P. Gummadi

Abstract

Trust plays a fundamental role in the adoption of technology by society. Potential consumers tend to avoid a particular technology whenever they feel suspicious about its ability to cope with their security demands. Such a loss of trust could occur in important computing platforms, namely cloud, enterprise, and mobile platforms. In this thesis, we aim to improve trust in these platforms by (i) enhancing their security mechanisms, and (ii) giving their users guarantees that these mechanisms are in place.

To realize both these goals, we propose several novel systems. For cloud platforms, we present Excalibur, a system that enables building trusted cloud services. Such services give cloud customers the ability to process data privately in the cloud, and to attest that the respective data protection mechanisms are deployed. Attestation is made possible by the use of trusted computing hardware placed on the cloud nodes. For enterprise platforms, we propose an OS security model—the *broker security model*—aimed at providing information security against a negligent or malicious system administrator while letting him retain most of the flexibility to manage the OS. We demonstrate the effectiveness of this model by building BrokULOS, a proof-of-concept instantiation of this model for Linux. For mobile platforms, we present the Trusted Language Runtime (TLR), a software system for hosting mobile apps with stringent security needs (e.g., e-wallet). The TLR leverages ARM TrustZone technology to protect mobile apps from OS security breaches.

Kurzdarstellung

Für die gesellschaftliche Akzeptanz von Technologie spielt Vertrauen eine entscheidende Rolle. Wichtige Rechnerplattformen erfüllen diesbezüglich die Anforderungen ihrer Nutzer jedoch nicht zufriedenstellend. Dies trifft insbesondere auf Cloud-, Unternehmens- und Mobilplattformen zu. In dieser Arbeit setzen wir uns zum Ziel, das Vertrauen in diese Plattformen zu stärken, indem wir (1) ihre Sicherheitsmechanismen verbessern sowie (2) garantieren, dass diese Sicherheitsmechanismen aktiv sind.

Zu diesem Zweck schlagen wir mehrere neuartige Systeme vor. Für Cloud-Plattformen präsentieren wir Excalibur, welches das Erstellen von *vertrauenswürdigen Cloud-Diensten* ermöglicht. Diese Cloud-Dienste erlauben es den Benutzern, ihre Daten in der Cloud vertraulich zu verarbeiten und sich darüber hinaus den Einsatz entsprechender Schutzvorkehrungen bescheinigen zu lassen. Eine solche Attestierung geschieht mit Hilfe von Trusted Computing Hardware auf den Cloud-Servern.

Für Unternehmensplattformen stellen wir ein Sicherheitsmodell auf Betriebssystemebene vor—das *Broker Security Model*. Es zielt darauf ab, Informationssicherheit trotz fahrlässigem oder böswilligem Systemadministrator zu gewährleisten, ohne diesen bei seinen Administrationsaufgaben stark einzuschränken. Wir demonstrieren die Leistungsfähigkeit dieses Modells mit BrokULOS, einer Prototypimplementierung für Linux.

Für Mobilplattformen stellen wir die Trusted Language Runtime (TLR) vor, ein Softwaresystem zum Hosting von mobilen Anwendungen mit strikten Sicherheitsanforderungen (z.B. elektronische Bezahlungsfunktionen). TLR nutzt die ARM TrustZone-Technologie um mobile Anwendungen vor Sicherheitslücken im Betriebssystem selbst zu schützen.

Acknowledgements

The time working on my Ph.D. was an unforgettable experience. I am indebted to many people. First and foremost, I am grateful to my advisor Rodrigo Rodrigues. To work with him has been a real pleasure and a fantastic learning experience. His permanent support, warm encouragement, and thoughtful guidance have been constant in this journey, and were essential to bringing this work to the light of the day.

In addition to the close collaboration with my advisor, this thesis is the fruit of teamwork with other people. I would like to thank my collaborators for their contributions in the multiple projects that are part of this thesis: Bryan Ford, Krishna Gummadi, Himanshu Raj, Stefan Saroiu, and Alec Wolman. I am very privileged and proud to have worked with each of them.

I would like to thank the MPI-SWS faculty for their high scientific standards and strong work ethic. I am grateful to Krishna Gummadi, who inspired me to always seek relevant research problems and incentivized me to pursue my own ideas. I would also like to thank Peter Druschel. The discussions with him were extremely important in helping to disentangle some hard problems, and his feedback on my work has always been very constructive in the course of my Ph.D. career.

To the staff and students at MPI-SWS, I would like to express my heartfelt gratitude. To Rose, who helped me improve my English language skills and provided valuable feedback on this thesis. To the IT staff, in particular to Carina Schmitt, who has been tremendously helpful on many occasions and always attended to my requests with promptness and care. To the administrative staff, namely Brigitta Hansen and Claudia Richter, for their tireless assistance with the logistics of living in Germany. And to the students and friends at MPI-SWS: the discussions and companionship with them all made this journey a unique and fascinating experience. I will never forget it.

Finally, I would like to thank my family and close friends. They provided me with the balance I needed to realize this task, gave me the strength to overcome the challenges that looked insurmountable, and pointed me to the positivity from where it is always possible to restart whenever things look dark. To them I dedicate this work and effort.

Contents

1	Introduction	1
1.1	The Importance of Trust in Technological Society	1
1.2	Trust Issues in Modern Computing Platforms	1
1.2.1	Cloud Platforms	2
1.2.2	Enterprise Platforms	2
1.2.3	Mobile Platforms	3
1.3	Improving Trust in Modern Computing Platforms	4
1.4	Contributions	4
1.5	Structure of this Thesis	6
I	Improving Trust in Cloud Platforms	9
2	Motivation and Related Work	11
2.1	Limitations of the Current Cloud Computing Stack	11
2.2	Cloud Attestation	13
2.3	Goals, Assumptions, and Threat Model	14
2.4	Brief Primer on the Trusted Platform Module	15
2.4.1	Trusted Computing Abstractions	15
2.4.2	Trusted Platform Module Primitives	17
2.5	Related Work on Improving Trust in the Cloud	18
2.5.1	Based on Trusted Hardware	18
2.5.2	Based on Cryptography	19
2.6	Summary	20
3	Towards Trusted Cloud Computing	21
3.1	Trusted Cloud Computing Platform	21
3.1.1	Architecture of a Typical VM Hosting Cloud Service	21
3.1.2	Requirements for TCCP	23
3.1.3	Insights Underlying TCCP	24
3.2	Concerns with TPM Usage in the Cloud Setting	27
3.3	Summary	29
4	Building Trusted Cloud Services with Excalibur	31
4.1	Design Goals	31
4.2	Excalibur Design	32
4.2.1	System Overview	32

Contents

4.2.2	The Policy-Sealed Data Abstraction	33
4.2.3	Cryptographic Enforcement of Policies	35
4.2.4	Securing the Monitor	36
4.2.5	Monitor Scalability and Fault Tolerance	37
4.3	Detailed Design	38
4.3.1	System Interfaces	38
4.3.2	Policy Specification	39
4.3.3	Excalibur Certificates	40
4.3.4	Excalibur Protocols	45
4.4	Implementation	48
4.5	Evaluation	49
4.5.1	Protocol Verification	49
4.5.2	Performance Evaluation	50
4.5.3	Cloud Compute Service	53
4.6	Summary	54
II	Improving Trust in Enterprise Platforms	57
5	Motivation and Related Work	59
5.1	The Problem of IT Mismanagement in Organizations	59
5.2	Hierarchical Separation of Administrator Roles	60
5.3	Goals, Assumptions, and Threat Model	61
5.4	Related Work on Improving Trust in Enterprise Platforms	62
5.4.1	Security Models	63
5.4.2	OS Isolation Techniques	63
5.4.3	Restriction of Administrator Privileges	65
5.4.4	Security Mechanisms of Commodity Operating Systems	66
5.5	Summary	67
6	Enhancing the OS Security against Mismanagement Threats	69
6.1	Broker Security Model	69
6.1.1	General Design	69
6.1.2	Methodology	71
6.2	OS Broker Functionality	71
6.3	Broker-enhanced OS Design	74
6.3.1	Architecture	74
6.3.2	Enforcing the Information Security Invariant	75
6.3.3	Enforcing the Identity Protection Invariant	77
6.3.4	Enforcing the System Integrity Invariant	78
6.4	Implementation	80
6.5	Evaluation	81
6.5.1	Security	81
6.5.2	Manageability	81

6.5.3	Compatibility	82
6.5.4	Performance	83
6.6	Discussion	83
6.7	Summary	84
 III Improving Trust in Mobile Platforms		87
7	Motivation and Related Work	89
7.1	Security Needs of Emerging Mobile Apps	89
7.2	Hosting Mobile Apps in Trusted Execution Environments	90
7.3	Goals, Assumptions, and Threat Model	91
7.4	Brief Primer on TrustZone and NetMF Technologies	92
7.4.1	ARM TrustZone Technology	92
7.4.2	Microsoft .Net Microframework	94
7.5	Related Work on Improving Trust on Mobile Platforms	96
7.6	Summary	97
8	Trusted Language Runtime: Enabling Trusted Applications on Smartphones	99
8.1	Overview of Trusted Language Runtime	99
8.1.1	Design Principles	99
8.1.2	High-level Design	100
8.1.3	Development Scenario	101
8.2	Design of Trusted Language Runtime	102
8.2.1	Internals of the TLR	102
8.2.2	Programming Model	104
8.2.3	Trustbox Management	107
8.2.4	Runtime Support	109
8.2.5	Cross-world Communication	111
8.2.6	Memory Management	113
8.2.7	System Boot	114
8.3	Implementation	114
8.3.1	Hardware Testbed	114
8.3.2	Software Implementation	115
8.4	Use Cases	115
8.4.1	One-time Passwords	116
8.4.2	User Authentication	117
8.4.3	Secure Mobile Transactions	119
8.4.4	Access Control to Sensitive Data	121
8.5	Evaluation	122
8.5.1	Performance	123
8.5.2	TCB Size	126
8.5.3	Programming Complexity	128
8.5.4	Security Analysis	129

Contents

8.6	Summary	129
9	Analysis and Limitations	131
9.1	A Unified Model for Trusted Computing Systems	131
9.2	Limitations of Trusted Computing Systems	133
9.2.1	Limitations Related to Trusted Software	133
9.2.2	Limitations Related to Trusted Computing Primitives	134
9.2.3	Limitations Related to Trusted Hardware	135
9.2.4	Limitations Related to Trusted Third Parties	137
10	Conclusions and Future Work	139
10.1	Conclusions	139
10.2	Directions for Future Research	140

List of Figures

2.1	Cloud computing layers.	12
2.2	Using trusted computing abstractions to provide DRM protection.	16
3.1	Architecture of a typical VM hosting cloud service.	22
3.2	“TPM benchmarks run against the Atmel v1.2 TPM in a Lenovo T60 laptop, the Broadcom v1.2 TPM in an HP dc5750, the Inneonv1.2 TPM in an AMD machine, and the Atmel v1.2 TPM (note that this is not the same as the Atmel TPM in the Lenovo T60 laptop) in the Intel TEP. Error bars indicate the standard deviation over 20 trials.” [MPP+08b]	27
4.1	Excalibur deployment. The dashed lines show the flow of policy-sealed data, and the solid lines represent interactions between clients and the monitor. The monitor checks the configuration of cloud nodes. After a one-time monitor attestation step, clients can seal data. Data can be unsealed only on nodes that satisfy the policy.	32
4.2	Example certificate tree and manifest. The certificates in light colored boxes form the <i>manifest</i> that validates the monitor’s authenticity and integrity.	39
4.3	Node attestation protocol.	45
4.4	Batch attestation example. The tree is built from 4 nonces. A summary for nonce n_{10} comprises its tag and the hashes in the path to the root.	46
4.5	Monitor attestation protocol.	47
4.6	Hook to intercept migration (from file <i>XendDomain.py</i>.) We redirect the state of the VM through a process that seals the data before it proceeds to the destination on socket <i>sock</i> (lines 1327-1330).	49
4.7	Performance of decryption key generation. Time to generate key as we vary the number of attributes (left), and throughput for 10 attributes as we vary the number of cores (right).	51
4.8	Performance overhead of sealing and unsealing data as a function of the complexity of the policy, with input data of constant size (1K bytes).	52
4.9	CPABE fraction in the performance overhead of sealing (left) and unsealing (right), varying the size of the input data.	53
4.10	Latency of VM operations in Xen. Encrypting the VM state accounts for the largest fraction of the overhead, while the execution time of CPABE is relatively small. Encryption runs AES with 256-bit key size.	54

List of Figures

5.1	Hierarchical separation of administrator privileges in two roles: <i>fully trusted</i> and <i>partially trusted</i>	61
5.2	Representative systems that can isolate the runtime state of users' computations from the administrator. Different systems enforce different isolation granularities: <i>virtual machine</i> , <i>process</i> , and <i>function</i> . The numbers in each diagram correspond to the protection rings found in Intel architectures, ranging from the most privileged (-1) to the least privileged (3) protection ring.	64
6.1	Software system under the broker security model.	70
6.2	Broker-enhanced OS architecture. The numbers in each layer correspond to the traditional protection rings, ranging from the most privileged (0) to the least privileged (3).	74
6.3	State transitions between account states: The user must explicitly accept that the account is valid before it can be used. In the active state, the administrator can temporarily disable the account or force the user to change authentication credentials. The resources of a deleted account can be released at a later point in time.	77
6.4	Performance of brokers when executed by the administrator: Covers representative brokers relative to package, account, group, module, and process management. The brokers for installing, getting, and removing packages use the <code>hello</code> package, which suffices for measuring the broker overhead for any package.	82
7.1	Architecture of a mobile platform featuring trusted execution environments.	91
7.2	Processor modes and hardware architecture of ARM-based device with TrustZone extensions.	93
7.3	Architecture of the .Net Microframework (NetMF).	95
8.1	High-level architecture of TLR.	100
8.2	Component diagram of the entire system with the TLR. The components of the same layer are colored with the same color.	102
8.3	Messages exchanged within and across the layers of the TLR software stack.	104
8.4	Code sample of a TLR application (written in C#).	105
8.5	Development workflow of a TLR application.	107
8.6	Details of the trustbox layer.	108
8.7	Details of the system layer.	109
8.8	Details of the trustzone layer.	112
8.9	State machine of a TLR call as implemented in the trustzone layer. Events in bold take place in the NW, and events in italic in the SW.	113
8.10	Execution time of trustlet methods from our use case prototypes.	124
8.11	Performance of our benchmark suite executed on the TLR and on Mono.	125
8.12	Minimum execution time of TLR primitives.	125

8.13	Performance of cross world method invocation varying the size of the method parameters.	126
8.14	Performance of seal and unseal primitives varying the size of sealed and unsealed data, respectively.	127
9.1	Key elements of a general trusted computing system: <i>trusted software</i> , <i>trusted hardware</i> , <i>trusted computing primitives</i> , and <i>trusted third parties</i> . .	131

List of Tables

4.1	Example of service attributes. In this case, EC2 supports two types of VM instances, two types of VMMs, and four availability zones (data-centers) in the US and Germany.	34
4.2	Example of a node configuration. This configuration contains the values for the attributes that characterize the hardware and software of a specific node N	34
4.3	Examples of policies. P_1 expresses version and VM instance type requirements, P_2 specifies a zone preference for one of two sites, and P_3 expresses a regional preference.	34
4.4	Excalibur service interface.	38
4.5	Certificate and manifest formats. A certificate $C_{Provider}^{Service}$ identifies the service, the attributes, and the certifiers. A certificate $C_{Certifier}^{Attribute}$ identifies a list of attributes of a service vouched for by a certifier. Certificates $C_{Certifier}^{Identity}$ and $C_{Certifier}^{Fingerprint}$ validate identities and fingerprints, respectively. Manifest M comprises certificates of service, attributes, and monitor identity and fingerprint. Square brackets indicate a list.	43
4.6	Performance overhead of sealing and unsealing data, varying the size of the input data.	53
6.1	Management tasks grouped into categories: Tasks are grouped by category. For each task we indicate the security invariants they violate: information security (IS), identity protection (IP), and system integrity (SI).	72
6.2	List of representative brokers grouped into categories: States each broker’s functionality and command name (in parenthesis).	76
8.1	Use Case 1: Online banking transfers.	116
8.2	Use Case 2: Mobile ticketing.	117
8.3	Use Case 3: Mobile payments.	119
8.4	Use Case 4: E-health application.	121
8.5	TCB size of the TLR, TrustVisor, and Mono+Linux setup.	127
8.6	Programming complexity of the use case prototypes measured in code size and number methods.	128

1 Introduction

1.1 The Importance of Trust in Technological Society

Today, computing technology permeates every aspect of modern society. Over the last 50 years, especially since the advent of the personal computer and the Internet, remarkable innovations in computing technology have been enthusiastically adopted by society. A wide range of hardware, software, and services have deeply affected the lives of individuals and organizations in all sorts of human activities, from entertainment (e.g., games) to mission-critical tasks (e.g., industry, health care, and finances).

This proliferation of technology was largely possible due to the web of trust that has been built between consumers and providers of technology. Since consumers do not generally have direct knowledge of the technology internals, their confidence about a particular product must be based on trust. Consumers' trust is built through the progressive accumulation of evidence in favor of a given technology, to the point where the risks of failure to meet the customers' expectations become tolerable. A good example of this process is online banking. Online banking became prevalent due to multiple contributions in strengthening end-users' trust. These contributions included the development of security mechanisms (e.g., cryptography, anti-virus, browser security, security protocols) and the coverage of user losses by banks and insurance companies in case of security breaches (e.g., phishing, identity theft).

History has also shown that customers' trust is fragile and can be easily eroded due to misjudged moves by the technology providers or by the limitations of the technology itself. Episodes where Facebook and Instagram have made their privacy policies more permissive were badly received by the public and the popularity of these services was immediately affected [ins, fbi]. Similarly, the loss of customers' data by Amazon S3 represented a significant blow to the credibility of cloud computing [amad]. To prevent a slowdown in the adoption of technology, it is then crucial that the providers of technology continue to be diligent in maintaining their customers' trust.

1.2 Trust Issues in Modern Computing Platforms

We highlight three important computing platforms that have not entirely been able to cope with the security expectations of their respective consumers, namely *cloud*, *enterprise*, and *mobile* platforms.

1.2.1 Cloud Platforms

Cloud platforms are one clear scenario where building trust is as important as it is challenging. Cloud computing follows an outsourcing model where cloud providers monetize their datacenter infrastructure by providing *cloud services* such as Amazon S3 [Amac] and Amazon EC2 [Amab]. Customers can then offload data hosting and computation to the cloud by paying for the resources consumed.

Since the customers pay for these cloud services, they expect their data to be handled properly in the cloud. As real world incidents have showed [tmo], failure by the cloud providers to handle customers' data, e.g., by leaking or losing data, could be catastrophic for customers and deeply affect the reputation of cloud providers. For this reason, cloud providers try to build customers' trust by making their systems reliable, for example, securing their premises, recruiting skilled engineers, and complying with best practices [Amaa].

However, despite the best efforts of the cloud providers, customers have expressed several concerns about the cloud. First, a lack of transparency is prevalent. Mostly due to security and business concerns, cloud providers tend to be secretive about the internals of their cloud infrastructures. This lack of transparency raises numerous doubts in customers' minds. Customers don't know, for example, who can access the data, who manages the cloud infrastructure, what software is really installed, how their data is being used, or on which locations (and jurisdictions) the data will be stored.

Second, current cloud platforms are prone to mismanagement threats. The cloud administrators, who are responsible for installing, configuring, and operating this software, could alter the behavior of a cloud service by reinstalling, reconfiguring, or manipulating the software of the cloud nodes. When performed by a negligent or a malicious cloud administrator, such activities could result in the leakage, corruption, or loss of customer data. Presently, this lack of guarantee about the behavior of cloud services deters many organizations from using the cloud for security sensitive tasks [ENI09a].

1.2.2 Enterprise Platforms

Trust issues could also arise in the context of enterprise environments. Many organizations use in-house *enterprise platforms* for storing and processing security sensitive data. By enterprise platforms we refer to the cluster and server infrastructures that constitute the IT backbone of an organization. These platforms take care of security sensitive data relevant not only to the organization itself, but also to external users, e.g., when hosting social networks sites, search engines, and shopping services.

In order for organizations to make sure that their enterprise platforms operate correctly, they must entirely trust their system administrators to do their jobs properly. In general, however, building trust in system administrators is not easy. System administrators are responsible for maintaining enterprise platforms, i.e., managing their software, resources, and the user data located therein. Because even small mistakes when performing these tasks could result in serious security breaches, system administrators must be highly trustworthy employees. While in small organizations administrators can be

closely scrutinized, in large organizations assessing the competence and tracing the behavior of individual employees is harder. Consequently, large organizations are more prone to security breaches due to negligent or malicious administrator activity.

In the current state of affairs, preventing mismanagement threats is not easy without significantly hindering the manageability of systems. Enterprise platforms typically run commodity operating systems (OSes), which require acquiring superuser privileges to perform most of the management tasks. While superuser privileges allow for the maximum flexibility in maintaining an OS, they could easily be abused in order to compromise sensitive user data. Existing defense techniques would either require deep changes to existing systems [BLP76, Bib77] or prevent administrators from performing most of their typical maintenance tasks [ZCCZ11]. Thus, it is time to rethink the design of commodity OSes so as to improve the security of enterprise platforms against administrator threats while preserving the system manageability.

1.2.3 Mobile Platforms

Lastly, we turn our attention to the mobile computing universe. Mobile platforms have witnessed an impressive boost in popularity over the last few years. A variety of mobile technologies became ubiquitous, such as laptops, netbooks, tablets, and smartphones.

As the mobile device market gained momentum, two interesting phenomena emerged. First, the impressive computing power of smartphones combined with the fact that they accompany their users everywhere prompted the emergence of a multi-million dollar mobile software industry. Thousands of mobile applications have been created by independent developers and distributed to users via online app stores [Goob]. Existing mobile apps offer their users a variety of services for photo sharing, password management, contacts management, and much more. Emerging applications promise to further enable payments in shops and vending machines and manage the health history of the smartphone owners—the so called *e-wallet* and *e-health* applications.

Another relevant change in the mobile sphere was the proliferation of malware. As the mobile applications started to process sensitive user data of high monetary value in the underworld (e.g., personal photos and location trails), spammers and identity thieves have increasingly deployed malware with the purpose of extracting that data. However, devising effective defense mechanisms against malware is far from trivial due to the complexity of the operating system and applications of mobile devices. In fact, the trusted computing base of mobile platforms is currently on par with that of applications running in desktops, opening an avenue for security breaches. As a result, today's smartphone platforms offer limited protections for processing security sensitive data, a fact that could erode users' trust and hinder the development of applications with stringent security requirements.

In summary, in cloud, enterprise, and mobile platforms, trust issues arise mostly due to technical limitations specific to each of the targeted computing platforms. Without addressing these limitations, users could decide to abandon a technology entirely (e.g., in cloud computing), be forced to use it with the associated risks due to the lack of

a better alternative (e.g., in the enterprise setting), or be deprived of interesting new applications (e.g., in the smartphone world).

1.3 Improving Trust in Modern Computing Platforms

In this thesis, we aim to strengthen users' trust in cloud, enterprise, and mobile computing platforms by building systems that can provide the following two key features:

1. **Enforce the security properties required by the users.** First, we aim to reinforce the protection of users' data and computations by enhancing the security of the computing platforms. The specific security properties to be implemented and the threat model under which they must be implemented are platform specific. In the cloud setting, we aim to prevent cloud administrators from inspecting or interfering with computations taking place in customers' virtual machines. In the enterprise environment, we want to enable administrators to maintain the operating systems without compromising the confidentiality and integrity of data located and processed in user accounts. In mobile environments, our goal is to develop mechanisms for protection of the mobile applications' state in the event of security breaches that could compromise the entire OS.
2. **Give users guarantees that the desired security properties are being enforced.** Second, because in most cases users do not have control over the computing platforms, even if a target platform enforces their desired security properties, users do not have the means to learn about the platform state and cannot tell whether or not it can be trusted. Therefore, it is fundamental to bridge this gap by giving users guarantees regarding the deployment of the mechanisms that enforce the desired security properties. To provide such guarantees, we leverage two techniques: *trusted computing hardware*, which provides online mechanisms for remote attestation of a platform's state, and *trusted certifiers*, which provide offline certification services.

Implementing this twofold strategy for cloud, enterprise, and mobile platforms raises new technical challenges, which we address with a set of novel contributions.

1.4 Contributions

The contributions of this thesis are as follows:

1. **The first cloud architecture that leverages trusted computing hardware for providing enhanced security in the cloud.** To address the trust issues in the cloud space, we present a cloud architecture named Trusted Cloud Computing Platform (TCCP). It consists of an Infrastructure-as-a-Service (IaaS) cloud service akin to Amazon EC2 that provides guarantees of confidentiality and integrity of customers' guest virtual machines from insider threats within the cloud. The key

insight underlying TCCP is a combination of a hardened virtualization layer that can host the guest VMs securely with a novel *cloud attestation* capability. Cloud attestation leverages commodity trusted computing hardware—Trusted Platform Module (TPM) [Gro06]—deployed on the cloud nodes to give customers guarantees that their virtual machines can execute only on the hardened virtualization layer. Although we illustrate cloud attestation with TCCP, this technique could be used more generally for building arbitrary trusted cloud services.

2. **A system that retrofits commodity trusted computing hardware into cloud infrastructures and provides simple primitives for building trusted cloud services.** Although TPMs alone could be used to implement trusted cloud services like TCCP, the developers of such services would face important challenges. In particular, without careful design, trusted cloud services could incur scalability bottlenecks, privacy breaches, and data management inflexibility. Such challenges emerge because TPMs have been developed for single node platforms and not for the multi-node cloud environment, which has unique requirements. To overcome the challenges of TPM usage in the cloud, we developed a system called Excalibur. Excalibur masks the intricacies of TPMs by (i) hiding the low-level TPM primitives from developers and (ii) offering developers a simple programming abstraction. This abstraction, named *policy-sealed data*, provides two primitives: *seal* and *unseal*. Seal enables customers to encrypt data to a user-defined policy before shipping it to the cloud, with the guarantee that the data can only be unsealed (i.e., decrypted) on the cloud nodes that satisfy the policy. The user-defined policy restricts the software and hardware configurations of cloud nodes according to the user preferences. Excalibur hides the low level details of the cloud, can cope with the data mobility needs within the cloud, and can scale massively. We demonstrated the practicality of Excalibur in Eucalyptus [NWG⁺], an open-source cloud platform.
3. **A novel OS security model and extensions for securing data and computations from mismanagement threats in commodity OSes.** In the context of enterprise platforms, we studied the problem of enabling an untrusted administrator to maintain a commodity OS while preserving the confidentiality and integrity of users' data and computations. Providing such protections is challenging because many tasks (e.g., creating user accounts, installing applications, or backing up user data) require granting the administrator superuser privileges, which give him direct access to users' data and computation state. To address this challenge, we propose a new set of guiding principles for OS design that we call the *broker security model*. Our model achieves a security-manageability trade-off by applying the principle of least privilege and prescribing the OS designer a methodology that (i) restricts administrator privileges by precluding inspection and modification of user data, and (ii) allows for the execution of necessary management tasks through the mediation of a layer of trusted programs—*brokers*—interposed between the management interface and system objects. Brokers provide data security at the

1 Introduction

user account granularity while enabling the administrator to perform the typical OS management tasks. To demonstrate the viability of this approach, we built BrokULOS, a Linux-based OS that suppresses superuser privileges and exposes a narrow management interface offered through a set of tailor-made brokers.

4. **A new system for protecting the execution state of security-sensitive applications on mobile platforms.** In the scope of mobile platforms, we present the Trusted Language Runtime (TLR), a system that provides a security environment for protecting the state of mobile applications for the .Net Framework in the event of an OS compromise. The TLR targets primarily smartphone devices where, despite the growing popularity of today's smartphones, they do not yet offer environments for building and running *trusted applications*, i.e., applications that require running security-sensitive logic in a trusted domain (e.g., for online banking). To facilitate the development of trusted applications, the TLR provides an intuitive programming model that enables developers to reason about the pieces of security-sensitive code called *trustlets*, and about the trusted domain environments, which are exposed to the programmer as a special sandbox object called *trustbox*. Then, at runtime, the TLR transparently confines trustlets' execution state to trusted domains, where it is kept safe from the reach of the OS. The TLR prevents bloating the TCB by making use of (i) ARM TrustZone technology [arm], which obviates the need for heavyweight hypervisors, and (ii) extending the .Net MicroFramework [net], which is a small footprint .Net language runtime for embedded and resource constrained devices. The TLR is easy to program because .Net offers the productivity benefits of modern high-level languages. We built a prototype of the system for an ARM emulator, and for a real hardware platform.

Some of the material in this thesis was previously published in a series of conference papers [SGR09, SRSW11, SRGS12, SRF12] or is under submission to a conference at the time of this writing [SRSW13].

1.5 Structure of this Thesis

The rest of this dissertation is divided into three parts, focusing on cloud, enterprise, and mobile platforms, respectively.

Part I introduces our contributions to improving trust in cloud platforms. In Chapter 2 we provide additional background and characterize the threat model in detail. Then, in Chapter 3, we present the TCCP cloud architecture and use it to highlight the challenges of TPM usage in the cloud; these challenges motivate the design of Excalibur. Chapter 4 presents the design of Excalibur, including the policy-sealed data abstraction, and the evaluation of the system.

Part II switches gears to focus on our contributions to enterprise platforms. Chapter 5 provides the background and related work in this space. Chapter 6 presents the Broker Security Model (BSM) and BrokULOS, a set of extensions for Linux that demonstrate the viability of BSM.

1.5 Structure of this Thesis

Part III focuses on improving trust in mobile platforms. Chapter 7 provides the background and related work, and Chapter 8 describes the design and implementation of the Trusted Language Runtime. Finally, Chapter 9 makes a comparative analysis of all systems developed in this thesis and discusses their limitations, and Chapter 10 concludes this dissertation by providing the main conclusions of this work and laying out the research directions we wish to pursue in the future.

Part I

Improving Trust in Cloud Platforms

2 Motivation and Related Work

In Part I of this thesis, we focus on the need to improve trust in cloud platforms. In fact, despite the benefits of cloud computing, the loss of control over data and computations constitutes a significant deterrent for potential cloud customers [ENI09a]. Existing cloud services fail to provide answers and guarantees for basic questions like: Who can access customer data and computations? Are they safe from cloud administrators? Are they safe from other cloud tenants?

We realize that many of these questions and uncertainties could be resolved if customers were assured that only the software they trust could be authorized to serve their requests. For example, if customer requests were served on a formally verified kernel that can isolate the domains of co-tenants and prevent access to computation state by the cloud administrators, the security guarantees offered by the service would be considerably stronger than if the job were done on a commodity hypervisor that does not offer such protections.

Based on this insight, our strategy to improve customers' trust is to enhance cloud services with a *cloud attestation* capability. Cloud attestation aims to give customers assurances that their requests are handled only by the cloud software that they trust. To provide this guarantee, this capability relies on commodity trusted computing hardware—Trusted Platform Module (TPM) [Gro06]—deployed on the cloud nodes to provide a reliable root of trust that is independent of the cloud nodes' software state. In Chapter 3, we illustrate how cloud attestation could be used in general for improving the security of cloud services akin to Amazon EC2 [Amab]. Then, in Chapter 4, we focus on the challenges of employing TPMs in cloud environments and present Excalibur, a system that helps overcome those challenges. Excalibur offers cloud providers a simple yet powerful primitive for building *trusted cloud services*, i.e., cloud services that take advantage of the cloud nodes' TPMs, while overcoming the limitations of TPM technology in the cloud.

In the rest of this chapter we present our motivation and related work in more detail. We start by providing an overview of the current cloud architecture and of its problems. We then introduce our idea—the notion of cloud attestation—aimed at addressing these problems, and clearly state our goals, assumptions, and threat model. Then, we provide a brief overview of the trusted computing technology, which we use to implement cloud attestation, and discuss the related work on improving trust in the cloud.

2.1 Limitations of the Current Cloud Computing Stack

To motivate the need for cloud attestation, we must first understand the risks that cloud customers incur in the current cloud computing model. A simplified model of existing

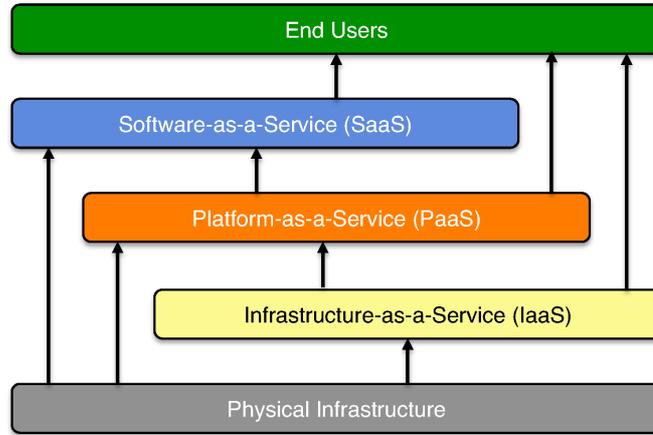


Figure 2.1: Cloud computing layers.

cloud services can be represented by the diagram in Figure 2.1. Despite the diversity and complexity of services and players that populate the cloud ecosystem, existing cloud services can be grouped according to the abstraction layer at which services are delivered to their respective clients:

- **Infrastructure-as-a-Service (IaaS)** includes the basic infrastructure services for virtual machine hosting (e.g., Amazon EC2) and data storage (e.g., Amazon S3). Operated by cloud providers like Amazon and Google these services run directly on a hardware infrastructure consisting of geographically dispersed datacenters, each of them hosting thousands of cloud nodes and other hardware elements. The software infrastructure that implements IaaS executes on the cloud nodes and consists of low-level software components, including a hypervisor or an operating system for virtual machine hosting or data storage services.
- **Platform-as-a-Service (PaaS)** sits on top of the physical infrastructure or IaaS. Similarly to IaaS, PaaS incorporates services for computing and storing data. However, these services are offered at a higher level of abstraction (e.g., databases, runtime and web app hosting) and are supported by a richer set of auxiliary services (e.g., message handling). Examples of PaaS services include Google AppEngine [Gooa] and Microsoft Azure [Azu]. PaaS services are typically implemented by middleware components that operate on top of the operating system and include execution runtimes (e.g., Java), frameworks, and database servers.
- **Service-as-a-Service (SaaS)** implement applications such as CRM, games, mail, portals, etc. SaaS services can be implemented on “bare metal”, on PaaS, or on IaaS (hosted in a virtual machine).

In all these cases, irrespectively of the abstraction layer at which services are offered, clients have limited awareness of and no guarantees about the service behavior.

Firstly, the implementation of the services is kept private by the service provider. Service providers tend to reveal only the interfaces and a high level description of the services. Hiding the low level implementation details aims to improve security from external attacks and preserve the competitive advantage over the provider’s peers. Secondly, the behavior of services is not guaranteed to be stable over time, even if the service logic is correctly implemented. Instability is possible because the cloud software stack is fully reconfigurable. For example, by changing the configuration of a service, a cloud operator could entirely alter its expected behavior. Such a fact could result in security breaches, service disruption, performance degradation, and functional deviation. This lack of assurances by today’s cloud architectures are unacceptable to clients with stringent security demands [ENI09b].

2.2 Cloud Attestation

To address the limitations of the current cloud stack, we propose to extend the cloud architecture with *cloud attestation*. We define cloud attestation as a mechanism that ensures that client requests can only be served on the cloud nodes whose software state is deemed trusted by the clients. The service provider retains the right and the freedom to implement the software of a service, but clients can now know and decide whether that software configuration is satisfactory even before using a cloud service.

Cloud attestation is a general mechanism for bootstrapping trust in the cloud. Cloud providers can specify the behavior of their services as an arbitrary state machine, enabling them to tailor the software configurations of the cloud nodes to enforce properties demanded by the clients. For example, a cloud provider could build an IaaS hosting service based on a formally verified microkernel such as seL4 [KEH⁺09], and leverage cloud attestation to give clients the guarantee that their requests will be served exclusively by the cloud nodes running seL4. Likewise, cloud services could be enhanced to satisfy certain requirements in terms of security (e.g., isolation properties), functionality (e.g., satisfying certain regulations), and performance (e.g., implementing specific optimizations).

To reliably convey the software configuration of cloud nodes to clients, we base cloud attestation upon *trusted computing hardware*. Namely, the cloud nodes are equipped with commodity trusted computing hardware, namely Trusted Platform Module (TPM) chips, which constitute the root of trust for cloud attestation. TPMs allow for checking the software state of the cloud nodes and making this information available to the clients. This capability is possible because TPMs’ primitives enable tracking the software state of a computer and reporting that state to a remote party. Since TPMs are inexpensive and increasingly available on server blades, deploying TPMs on a cloud infrastructure would require modest or no additional investment by the cloud provider.

There are, however, some concerns that need to be addressed in order to make this technique practical. One potential concern for cloud providers is related to how many details about the cloud would be revealed to the public by providing cloud attestation. To limit the amount of information that is made public, cloud attestation must only

convey the software configuration of the cloud nodes to clients in a form that is both meaningful for clients and not compromising for providers. Another potential concern is whether cloud attestation will overly complicate the development and maintenance of cloud services, or affect the scalability and fault tolerance of services. Next, we draw our plans to make cloud attestation practical.

2.3 Goals, Assumptions, and Threat Model

In Part I of this thesis we focus on (i) illustrating the benefits of cloud attestation, and (ii) addressing the potential concerns that this technique could raise. To illustrate the potential benefits of cloud attestation, we focus exclusively on IaaS services, which constitute the bedrock of the cloud computing stack, and present an architecture of an IaaS service that provides data security from malicious administrators (see Chapter 3). To address the concerns of cloud attestation, we built Excalibur. Excalibur is a system that helps retrofit the cloud infrastructure with TPMs, and provides a key high-level primitive—*policy-sealed data*—for developing and managing trusted cloud services (see Chapter 4). Extending cloud attestation to the cloud stack’s upper layers raises additional challenges, which we plan to explore in the future (see Chapter 10). Next, we clarify our assumptions and threat model.

We differentiate between trusted and untrusted software platforms; the former are approved by the clients, whereas the latter are not. We assume that the trusted software platforms are correctly implemented and have the capability to protect volatile key material generated by Excalibur. Since our focus in Part I is on providing a cloud attestation capability, we are not concerned with securing the software platforms themselves. Such protections would require sanitizing the management interface exposed to the cloud administrators to prevent leakage or corruption of data (e.g., direct memory inspection). To address these complementary goals, the developers of the trusted software could make use of existing systems and hardening techniques presented in previous research [MSWB09, KEH⁺09, ZCCZ11, HHF⁺05] and in Part II of this thesis. Regarding the untrusted software platforms, we make no assumptions whatsoever.

We assume that all cloud nodes are equipped with TPMs and that the hardware is correctly implemented. In addition, we assume that the physical integrity of the cloud nodes is protected. It is often the case that in modern datacenters the physical access to the cloud nodes is highly restricted. In fact, most of the management activity is performed from remote sites, including installing software, monitoring systems’ activity, and power cycling the cloud nodes. In some cases, such as in container-based datacenters [Ham07], physical access is entirely prohibited.

Regarding the threat model, cloud attestation must be robust against a malicious agent that operates from within the cloud. An attacker must not be able to fool the cloud clients into thinking that cloud nodes execute a particular software when in reality they execute a different one. We model the attacker’s capabilities as those of a disgruntled cloud administrator with the privileges to manage the cloud software remotely: he can reboot any cloud node, access its local disk after rebooting, reinstall the software, and

eavesdrop the network. He can install an operating system or hypervisor that he controls on any cloud node, allowing him to have full control over a cloud node's state. However, installing a trusted software platform on a cloud node restricts the attacker's capabilities to the privileges that that software grants the administrator. For example, on a cloud node booting the seL4 [KEH⁺09] microkernel, an attacker could not control the OS kernel nor the system services. Finally, the attacker cannot launch physical attacks that could compromise the TPMs, because he has only remote access to the nodes, and he is powerless to violate the integrity of trusted software platforms.

Note that, although we model the attacker as a disgruntled cloud administrator, our threat model covers a broad range of threats. By protecting against a malicious cloud administrator, cloud attestation offers defenses against accidental or negligent activity by the cloud administrators. In addition, this threat model also covers attacks that escalate administrator privileges on untrusted software platforms stemming from malware infection or from external attackers. Next, we provide some background on TPM technology and discuss the related work.

2.4 Brief Primer on the Trusted Platform Module

As mentioned above, a key building block for cloud attestation is the Trusted Platform Module (TPM). To better understand how this technology works, we provide some minimal background. First, we introduce the main abstractions implemented by TPMs, and then describe the most relevant implementation details.

2.4.1 Trusted Computing Abstractions

The Trusted Platform Module (TPM) [Gro06] is the most popular and widespread instance of trusted computing hardware technology. The primary goal of trusted computing hardware is to implement a set of trusted computing abstractions, which allow for bootstrapping trust in a single computer [PMP10]. These abstractions enable a remote party to 1) reliably determine the bootstrap execution state of a computer, and 2) restrict data access on that computer to a software execution state trusted by the remote party. Trusted computing abstractions are important in cloud attestation because they will be used as the fundamental operations for building trust in the multi-node cloud environment. To better understand the role of these abstractions, we first introduce their semantics and then use an example application to illustrate how they work.

There are typically four main abstractions that the trusted hardware is expected to implement:

- **Strong identity:** Strong identity enables the computer to be uniquely identified without having to trust the OS or the software running on the computer.
- **Trusted boot:** Trusted boot produces a unique fingerprint of the software platform running on the computer; the fingerprint consists of hashes of software platform components (e.g., BIOS, firmware controlling the computer's devices, boot-loader, OS) computed at boot time.

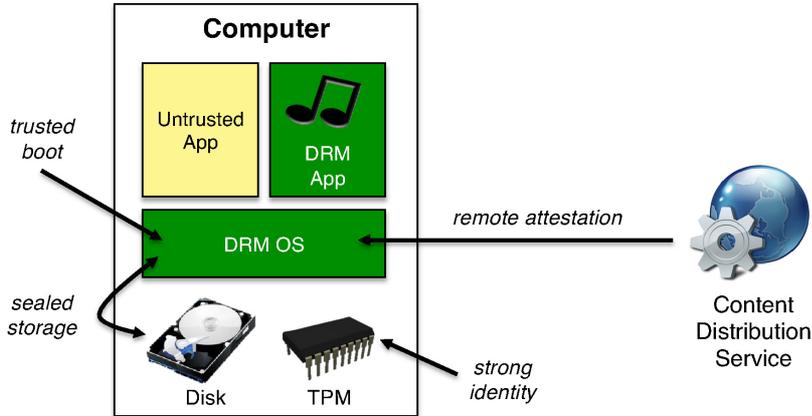


Figure 2.2: Using trusted computing abstractions to provide DRM protection.

- **Remote attestation:** The software fingerprint can be securely reported to a remote party using a *remote attestation* protocol; this protocol lets the remote party authenticate both the computer and the software platform so it can assess whether the computer is trustworthy, e.g., if it is a trusted platform that is designed to protect the confidentiality and integrity of data [Mic, HHF⁺05].
- **Sealed storage:** Sealed storage allows the system to protect persistent secrets (e.g., encryption keys) from an attacker with the ability to reboot the machine and install a malicious OS that can inspect the disk; the secrets are encrypted so that they can be decrypted only by the same computer running the trusted software platform specified upon encryption.

These abstractions can then be used by a particular software platform to provide specific guarantees. Consider for example how they could be used for building an OS with Digital Rights Management (DRM) protection in desktops (see Figure 2.2). The goal of DRM is to prevent illegal retrieval and distribution of copyrighted content, such as music, movies, and software. The key challenge of implementing DRM is that the protections in the operating system (OS) and applications for checking the authenticity of the content can be subverted and allow for the direct access to the content. Bypassing these protections could be done by tampering with the binaries of OS and applications, and then boot the modified versions on the computer.

Thwarting copyright infringement could be achieved using trusted computing hardware located on the consumer’s platform and leveraging trusted computing abstractions. Intuitively, the idea is to give content distributors the guarantee that the binary of the OS and applications have not been tampered with. If this property holds, it is safe to ship the content to the consumer’s platform. This verification could be done as follows. Before shipping the content, the content distribution services execute a *remote attestation* protocol which obtains a remote fingerprint of the target computer. (The

strong identity could be (optionally) used to authenticate the remote attestation signatures issued by the consumer platform.) If the fingerprint differs from the trusted software's, then the trusted software binary has been modified or replaced before boot. Thus, the platform cannot be trusted, and the content delivery is aborted. Otherwise, the consumer platform is trusted and the service proceeds with sending the content.

To protect the content at the consumer's end, the OS must prevent the content from being leaked from memory and from disk. For this last purpose, the OS uses *sealed storage* before storing the content on disk so that, if the platform reboots to a different OS, the content remains bound to the trusted software and is inaccessible to the untrusted OS. In order for remote attestation and sealed storage to work, *trusted boot* must be enabled, so that the fingerprint can be generated upon boot.

This example helps to show how trusted computing abstractions provide the basic support for enabling a remote party to bind sensitive data to trusted software on a single-node platform. (In fact, thwarting copyright infringement was one of the motivations that drove the development of trusted computing hardware and TPMs.) In our work, we borrow and develop this idea to provide analogous guarantees in multi-node cloud environments.

2.4.2 Trusted Platform Module Primitives

The TPM's main goal is to implement the trusted computing abstractions described above. In reality, the functionality of TPM exceeds this scope. The TPM could be seen as a non-programmable cryptographic library offering OS and application developers a large number of cryptographic primitives (107 commands in total for TPM v1.2), ranging from random number generation, cryptographic key generation and management, monotonic counters, and data encryption / decryption. In the context of this work, however, only a few subset of primitives are relevant to us, namely those that implement the trusted computing abstractions:

- To provide a strong identity, the TPM uses an Attestation Identity Key (AIK), a cryptographic public key generated inside the TPM; its private key never leaves the TPM in plaintext and can only be used for issuing digital signatures in the TPM.
- To track the hash values that constitute a fingerprint, the TPM uses special registers called Platform Configuration Registers (PCRs). Whenever a reboot occurs, the PCRs are reset and updated with new hash values; the content of the PCRs constitutes the fingerprint of the software platform booting on the machine.
- To perform remote attestation, the TPM can issue a *quote*, which includes the PCR values signed by the TPM with an AIK.
- For sealed storage, the TPM offers two primitives, called *seal* and *unseal*, to encrypt and decrypt secrets, respectively. Seal encrypts the input data and binds it to the current set of PCR values. Unseal validates the identity and fingerprint of the software platform before decrypting sealed data.

In the context of this work, we use these TPM primitives to provide cloud attestation support. In particular we use them to illustrate the design of a trusted cloud service (Chapter 3), and to build Excalibur (Chapter 4). But before presenting these issues in more detail, we discuss some relevant work on improving trust in the cloud.

2.5 Related Work on Improving Trust in the Cloud

We structure our survey on related work in two categories: techniques based on trusted hardware, and techniques purely based on cryptography.

2.5.1 Based on Trusted Hardware

To the best of our knowledge, we introduced the concept of leveraging trusted hardware to improve customers' trust in the cloud [SGR09]. Since our initial proposal, a stream of research has been produced on this topic. To review the work in this area, we first describe the relevant work in trusted computing in general, and then discuss the existing research on applying trusted computing in the context of cloud services.

Over the past several years, there has been considerable work on trusted computing [PMP10]. Most of this work targets single computers with the goal of enforcing application runtime protection [GPC⁺03, HHF⁺05, MPP⁺08a, MLQ⁺10, LTM⁺00], virtualizing trusted computing hardware [BCG⁺06], and devising remote attestation solutions based on both software [SPvDK04, HCF04] and hardware [SZJvD04, SJZvD, SPD05, BCC04, JSS, SS04]. Other work, focusing on distributed environments, provides integrity protection on shared testbeds [CHER10] or distributed mandatory access control [MJB⁺06]. More recently, trusted computing primitives have been adapted to mobile scenarios to provide increased assurances about the authenticity of data generated by sensor-equipped smartphones [LSWR12]. Our work concentrates on the specific challenges of cloud computing environments, which fall outside the scope of these prior efforts.

Excalibur shares some ideas with property-based attestation [SS04], whose goal is to make hash-based software fingerprints more meaningful to humans. Like Excalibur, property-based attestation maps low-level fingerprints to high level attributes (properties) and relies on a monitor (controller) to perform this mapping. However, this prior work offers an abstract model without an associated system. Moreover, Excalibur extends this work by proposing new trusted computing primitives.

Nexus [SdBR⁺11], a new operating system for trustworthy computing, introduces active attestation, which allows attesting a program's application-specific runtime properties and supports access control policies per application. Both Nexus policies and policy-sealed data can bind data based on attributes. However, Nexus and Excalibur target complementary problems: Nexus policies are tied to nodes running Nexus and restrict how the applications can access the data; Excalibur policies focus on multi-node settings and restrict how the cloud nodes, possibly running various software platforms, can access the data. Thus, Nexus could be a good candidate to use as an attribute in an Excalibur policy.

2.5 Related Work on Improving Trust in the Cloud

The work by Schiffman et al. [SMV⁺10] aims to improve the transparency of IaaS cloud services by providing customers with integrity proofs of their VMs and underlying VMMs. Like Excalibur, a central component, called cloud verifier (CV), mediates attestations of nodes and uses high-level properties (attributes) for reasoning about node configurations. However, the scope of this work is narrower than ours: while the CV provides only integrity proofs, Excalibur builds on these proofs to enforce policy-sealed data, which is a general, data-centric abstraction for protecting customer data in the cloud. In addition, the CV administrator is assumed to be trustworthy, representing a weaker threat model; in our view, this assumption does not address an important class of problems that occur in cloud services today. Finally, their system does not address the shortcomings of sealed storage TPM primitives, which could raise concerns of data management inflexibility and isolation crippling if these primitives need to be used by cloud services to secure persistent data.

Multiple software systems have been proposed to increase the security of sensitive data in the cloud. At the OS layer, hypervisors and OSES can protect the confidentiality and integrity of data using isolation [MLQ⁺10, KEH⁺09, ZCCZ11, RRT⁺11] or information flow control [VEK⁺07] techniques. At the middleware layer, a range of frameworks for building Web offer their users strict control over data remotely placed at the provider site [KMC11], enable controlled sharing of sensitive data using differential privacy [RSK⁺10], and provide general-purpose encapsulation mechanisms for data [MAF⁺11]. These proposals are complementary to our work: despite their potential to increase security and control over data in the cloud, these proposals lack a scalable mechanism for bootstrapping trust in the multi-node cloud environment. By combining these platforms with Excalibur, cloud providers could build new trusted cloud services.

2.5.2 Based on Cryptography

The most common alternative to using trusted hardware is based on cryptography. The idea is to protect the secrecy of customers' data by keeping it permanently encrypted while the data is hosted in the cloud: the data is encrypted before being shipped by the users to the cloud and can only be decrypted and retrieved at the customers' end.

The main strength of cryptography-based when compared to trusted hardware-based approaches is that no component needs to be trusted at the cloud provider's end in order to provide data confidentiality. However, pure cryptography-based techniques have significant limitations to provide secrecy protection if the data has to be computed in non-trivial ways on the cloud (i.e., other than replicating, comparing, and deleting ciphertext). Recently, a fully homomorphic scheme has been proposed [Gen09], which allows for arbitrary computations over encrypted data. However, this proposal constitutes a theoretical result for which an efficient practical implementation is yet to be discovered.

To support computations over encrypted data, some work has used more mainstream cryptography by making a trade-off between efficiency and functionality. One remarkable example is CryptDB [PRZB11], which supports queries of encrypted databases to some degree. This is possible by cleverly encrypting the databases with various crypto-

2 Motivation and Related Work

graphic schemes, each of them is able to support a subset of operations over encrypted data. While CryptDB constitutes a significant step towards supporting efficient secure computations, it still exhibits some limitations. Firstly, there are restrictions to the database queries that can be issued. Secondly, weaker cryptographic schemes are used, which degrades the overall security of the system. Given the difficulty of performing secure computations over encrypted data, it is not surprising that many systems using cryptography provide security for only use cases where data does not need to be computed. This is the case of secure storage services for the cloud [BCQ⁺11] or storage-intensive cloud applications [PKZ11].

In summary, we can say that the two main techniques for improving trust in the cloud offer different and complementary trade-offs. While the cryptography-based approaches can provide secrecy protection without requiring any trusted components in the cloud provider, trusted hardware-based approaches depend on the correctness of trusted components, but provide full support for efficient and arbitrary computations. In this thesis, we focus on the latter.

2.6 Summary

In this chapter we introduced cloud attestation, which constitutes our approach for improving customers' trust in the cloud. Cloud attestation is based on trusted hardware deployed on the cloud nodes and enables customers to bind their data and computations to software platforms they trust. After clarifying our goals, assumptions, and threat model, we provided some background on TPMs, the trusted hardware we use for cloud attestation, and discussed the related work on improving trust in the cloud. In the following chapters, we show how cloud attestation enables the design of cloud services that can provide privacy guarantees for customer computations, and discuss the challenges of using TPM technology in cloud attestation (Chapter 3). We then present Excalibur, a system that overcomes these challenges and assists cloud providers in building services with a cloud attestation capability (Chapter 4).

3 Towards Trusted Cloud Computing

In this chapter, we present a new cloud computing architecture for building *trusted cloud services*, i.e., cloud services that leverage cloud attestation in their design in order to give clients specific guarantees. We illustrate the potential benefits of such services by providing a concrete example of a trusted cloud service, called the Trusted Cloud Computing Platform. We use the same example to discuss the limitations of TPM technology in the cloud setting, limitations that motivate the design of Excalibur.

3.1 Trusted Cloud Computing Platform

The Trusted Cloud Computing Platform (TCCP) [SGR09] aims to provide a virtual machine (VM) hosting service with guarantees of secrecy and integrity protection of the VMs' state in the cloud. Before presenting the design of TCCP, we start by describing the internals of a typical IaaS VM hosting cloud service, and then present the additional security guarantees we aim to achieve with TCCP.

3.1.1 Architecture of a Typical VM Hosting Cloud Service

A typical VM hosting cloud service provides a functionality akin to Amazon EC2 [Amab]. Customers can rent instances of virtual machines, hosted in the cloud infrastructure and for which customers pay a price that depends on the time and the resources allocated. VMs are created from a VM image (VMI), which customers can select from a public repository provided by the service or upload to the cloud themselves. The cloud service is responsible for managing the resources of VMs and for securing the VMs' execution states, namely from interference by other tenants' VMs co-located on the same physical machine.

Figure 3.1 illustrates the architecture of a VM hosting cloud service. The components shown in the figure are, in reality, a simplification of a real world deployment. Nevertheless, they include the components that we find in the Eucalyptus [NWG⁺] open source cloud platform. The service is hosted in multiple clusters of cloud nodes. The bulk of the clusters are responsible for hosting the guest VMs. Internally, a cloud node runs a virtual machine monitor (VMM), which is responsible for managing the lifecycle of the guest VMs residing on that cloud node. The VMM controls the memory, CPU, network, and disk resources used by each VM, and sets the security policies of each VM.

In addition to the cloud nodes allocated to VM hosting, we find specialized internal services deployed on different clusters. There are three main such internal services: the *cloud manager*, the *VMI repository*, and the *VM repository*. The cloud manager is responsible for coordinating the customer VMs in the service. It manages customer

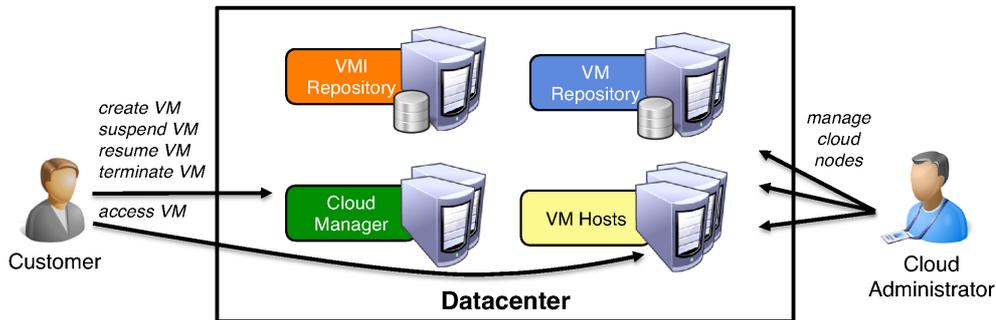


Figure 3.1: Architecture of a typical VM hosting cloud service.

IDs, serves the requests by the customers for managing the lifecycle of their VMs (e.g., create, terminate, suspend VMs), provides information about the VM images registered in the cloud service, reports billing information to customers, and provides a backend interface for monitoring the service. The VMI repository contains all the VM images supported by the service, each of them featuring different software configurations (e.g., OS, applications). The VM repository contains the images of VMs' execution states that have been suspended by the customers. These images can be resumed later on and their execution continued.

The VM hosting service allows customers and cloud administrators to interact with it using a frontend and a backend interface. We highlight the most relevant operations, which are those related to the management of VMs, where the sensitive computation state resides. The frontend includes operations that allow customers to *create*, *suspend*, *resume*, and *terminate* a VM. The backend includes, in addition, operations that enable the cloud administrators to *migrate* customers' VMs across cloud nodes. The migration operation is important for load balancing. These operations are coordinated by the cloud manager and work as follows.

VM creation. To create a VM, a customer uses an authenticated connection to issue a request to the cloud manager, indicating the characteristics of the VM (e.g., CPU speed, memory size) and the VMI that should be instantiated. The cloud manager validates the customer's identity and the other parameters, and designates a candidate cloud node for hosting the VM instance. Which cloud node is chosen depends on the internal policy for managing the resources and on the current resource allocation. The cloud manager then instructs the cloud node's VMM to create the VM instance. The VMM retrieves the VMI from the VMI repository, creates the VM, and boots the VM. During this process the VMM injects the customer's public key into the local VMI replica so that the customer can login to the VM once the VM boots. The customer can learn the booting status of the VM by consulting the cloud manager. Once the VM is up and running, the customer can now login over an SSH connection and perform his desired computations.

VM suspension. To suspend a VM, the customer instructs the cloud manager accordingly, which in turn forwards the request to the VMM of the cloud node where the VM is located. The VMM suspends the VM by freezing its volatile and persistent state, bundles this state on a single file containing the VM image, and ships it to the VM repository. The VMM then informs the cloud manager that the operation has finished, enabling the customer to learn about the status of the operation.

VM resumption. To resume the VM, the customer sends a request to the cloud manager. The cloud manager selects a free cloud node, just like when creating a VM, and requests the VMM of the selected cloud node to retrieve the VM image from the VM repository. Once the VM image is locally available to the VMM, the VMM restores the VM state and resumes the execution of the VM. Next, the VMM notifies the cloud manager, which in turn makes this information available to the customer. The customer can then login to the VM as before suspension.

VM termination. To terminate a VM, the customer issues a request to the cloud manager, which forwards the termination request to the VMM where the VM is hosted. The VMM terminates the VM, releases all its resources, and then updates the cloud manager, which then informs the customer. The VM termination is carried out in the same way when requested by the customer or by the administrator.

VM migration. Lastly, to migrate a VM from the current cloud node to another cloud node, the administrator instructs the cloud manager by indicating the target VM and the designated destination cloud node. The cloud manager contacts the VMMs of both the source and the destination cloud nodes, and the VMMs initiate a VM migration protocol for transferring the VM state between the nodes. Once the transfer finishes, the cloud manager is notified of the status of the operation.

3.1.2 Requirements for TCCP

A careful look at the architecture of a typical VM hosting cloud service shows that, in the face of security outbreaks that fit into the threat model defined in Chapter 2, the security of customer VMs is precarious. For example, outbreaks stemming from misconduct of cloud administrators or exploits of the service components by external agents could compromise the confidentiality and the integrity of the customers' VMs. The attack surface is large and attacks could be made in multiple ways through any of the components of the architecture, such as the following:

- **VMM:** The VMM typically consists of a commodity hypervisor like Xen. Commodity hypervisors provide strict isolation between guest VMs, but offer no protection against the management domain, i.e., the privileged domain from where the VMs' resources are controlled (e.g., Dom0 in Xen). In other words, once a VM is instantiated on the cloud node, the hypervisor can protect the VM's state from co-resident VMs, but not from the administrator of the system. From the

3 Towards Trusted Cloud Computing

management domain, the administrator has full privileges to access the volatile (in memory) or the persistent (on disk) state of a VM. As a result, an attacker empowered with administrator privileges could access the computation state of a VM, including any sensitive information located in the VMs (e.g., private keys, personal information, financial data).

- **VMI and VM repositories:** These repositories contain the initial state of a VM right before instantiating or resuming a VM, respectively. An attacker with access privileges to these repositories could read or modify their content arbitrarily. Without mechanisms, such as encryption, that could prevent data inspection and detect modifications, sensitive state of the VMs could be compromised. For example, the program binaries of a VMI could be modified to implant malware in order to leak customers' secrets or corrupt the computations.
- **Cloud manager:** The cloud manager controls the authentication of the customers to the VMs and designates the cloud nodes for hosting the VMs. Compromising the cloud manager would allow an attacker, for example, to have direct login access to a VM. Upon creation, the attacker could replace the public key of the customer for the attacker's public key, and pass the attacker's key to the VMM of the hosting cloud node. The VMM would then inject the attacker's key into the VM, allowing the attacker to login to the VM. Compromising the cloud manager could even allow an attacker to divert the VM outside the cloud provider's premises. During the creation, migration, or resume stages, an attacker could designate any machine to host a customer's VM, thus making them vulnerable even to physical attacks.

The goal of the Trusted Cloud Computing Platform is to mitigate these threats by reinforcing the security of the VM hosting service. Specifically, TCCP aims to provide secrecy and integrity protection of the state of customers' VMs throughout the entire VM lifecycle. Next, we show how to provide such protections.

3.1.3 Insights Underlying TCCP

The insight behind the design of TCCP consists of addressing two complementary sub-problems: first, protect the VM state in the cloud, and then give customers guarantees that those protection mechanisms are in place.

Protecting the VM State

The first step is to protect the state of customers' VMs. In the standard architecture of a VM hosting cloud service (see Figure 3.1), the security of customer's VMs is dependent on a huge trusted computing base (TCB). The TCB includes the VMM on the nodes, the VMI and VM repositories, and the cloud manager. Furthermore, all these components are designed under the assumption of a fully trusted administrator. To protect the confidentiality and integrity of VMs, our approach is then to factor out as many components as we can from the TCB, and then harden the leftover TCB components to prevent the cloud administrator from overriding the TCB's security protections.

Thus, we can protect the states of VMs by combining two techniques. First, we leverage a *hardened VMM* (HVMM) (e.g., CloudVisor [ZCCZ11]) for protecting the runtime state of VMs in their running stage. The HVMM must isolate the VM from the management domain, which can be done using existing techniques [GPC⁺03, ZCCZ11, RRT⁺11]. Second, we extend the cloud service protocols in order to keep the VM state encrypted throughout the remaining stages of the VM lifecycle. This design allows us to reduce the TCB size considerably by obviating the need to trust the cloud manager, the VMI repository, and the VM repository: only the HVMM belongs to the TCB.

The TCCP must then implement a set of distributed protocols that can provide the following assurances:

- During VM creation, the customer must be guaranteed that: (i) the VMI corresponds to the VMI selected by the customer (and has not been modified or replaced), and (ii) the public key injected into the VMI is the customer's and not someone else's. To provide both guarantees, the HVMM must carry out some additional steps. Before booting a VM, it computes the digest of the VMI and creates a record of the public key to be injected into the VMI. Then, once the VM boots, the HVMM enables the customer to read the VMI digest and the record of the public key. If either of the elements has been corrupted, the VM instance is not trustworthy, and the customer can abort the creation of the VM. Otherwise the VM instance is reliable, and the HVMM ensures its protection during execution.
- For VM migration, it is important to make sure that the VM state is protected while in transit over the network until it reaches the destination cloud node, at which point the HVMM will provide for the security of the VM after it resumes execution. To secure the VM migration, the HVMM endpoints could simply establish a secure channel using standard SSL, and then proceed with the VM state transfer over the secure channel.
- Between VM suspend and resume operations, the VM state could be vulnerable to inspection and modification from the moment it leaves the source cloud node and moves to the VM repository after suspension, until it resumes execution in the destination cloud node after being transferred from the VM repository. To secure the VM state in the interim, the VM state could be encrypted and appended with integrity digests at the source, and then decrypted and its integrity validated at the destination. The cryptographic key used to encrypt and decrypt the VM state could be maintained by the customer and propagated to the source and to the destination without putting an additional burden on the customer.

At first sight, this design is effective at securing the state of a VM by keeping it either (i) unencrypted on an HVMM, or (ii) encrypted while transiting over the network or stored in a repository. However, the VM is not yet secure. To provide these guarantees, cloud attestation is necessary.

The Need for Cloud Attestation

A fundamental missing piece is the lack of guarantees that the hosting nodes can actually be trusted. In the protocols shown above, an attacker could still assign the VM state to a machine executing an untrusted software platform or to a machine located outside the cloud provider's premises. Such an assignment could be made by the cloud administrator, for example, through the system's management interface. If a cloud node is not executing a correct binary of the HVMM, then the cloud node can no longer be trusted to properly isolate the VM state from the cloud administrator. Similarly, if the hosting node is located outside the cloud provider's premises, an attacker could launch arbitrary physical attacks. In either case, the designated machine could no longer be trusted for hosting the VM.

The security protocols of TCCP must, therefore, include additional checks for guaranteeing that the cloud nodes can be trusted. This is precisely the role of *cloud attestation*. Cloud attestation consists of cryptographic protocols whose goal is to make sure that customer VMs can only execute on a machine (i) owned and deployed in the cloud provider's premises, and (ii) executing the HVMM. For this purpose, cloud attestation uses the TPMs installed on the cloud nodes. The TPM's AIK key helps authenticate the node by providing a strong node identity that can be compared against a certified list containing the AIK public keys of all the cloud nodes deployed in the cloud provider's premises. The TPM's PCR values (i.e., software fingerprint produced during trusted boot) help determine whether the cloud node is executing the HVMM software.

To implement cloud attestation, the protocols described in the section above must be extended in order to validate the AIK key and PCR values of cloud nodes. In particular, these checks must take place before a node is authorized to receive the state of a VM, namely in all operations that assign a VM to a cloud node: *create*, *migrate*, and *resume*. To implement these protocol extensions, we can use standard remote attestation and sealed storage primitives as follows:

- Remote attestation can be used in the create and in the migrate protocols to check the authenticity and software identity of the target node. In the first case, the software running at the customer end checks the cloud node upon the VM instantiation. In the second case, the hosting node checks the destination cloud node upon migration.
- Sealed storage can be used in the suspend and resume operations to make sure that the encrypted VM state produced upon suspend can only be decrypted upon resume if the target machine where resume is taking place is trusted. This is possible by leveraging the fact that the same TPM must be involved in both seal and unseal operations (see Sections 2.4.1 and 2.4.2). To enforce this behavior, before transmitting the encrypted VM state to the VM repository, the VMM of the hosting cloud node seals the encryption key, which can now be stored along with the encrypted VM state. The resume cloud node is then forced to unseal the encryption key in order to recover that key. This operation will only succeed if the cloud node meets the trust requirements stated above. Thus, if the cloud node is

3.2 Concerns with TPM Usage in the Cloud Setting

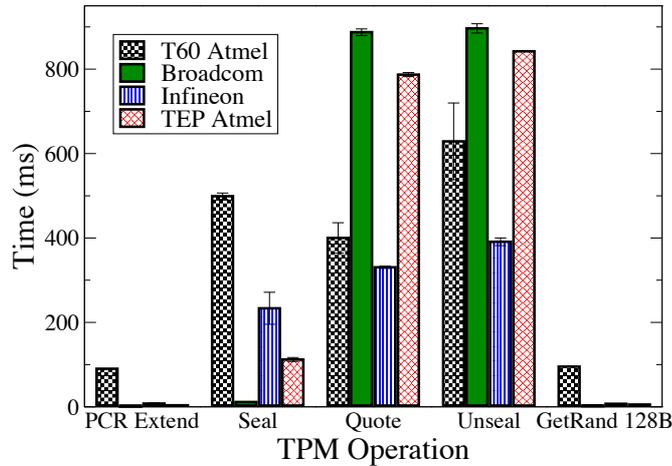


Figure 3.2: “TPM benchmarks run against the Atmel v1.2 TPM in a Lenovo T60 laptop, the Broadcom v1.2 TPM in an HP dc5750, the Infineon v1.2 TPM in an AMD machine, and the Atmel v1.2 TPM (note that this is not the same as the Atmel TPM in the Lenovo T60 laptop) in the Intel TEP. Error bars indicate the standard deviation over 20 trials.” [MPP⁺08b]

trusted, unseal succeeds, and the VMM is now able to proceed with decrypting the VM state and resuming the VM. Otherwise, unseal fails. Sealing has the benefit that the customer does not need to keep track of the encryption keys.

With this design, TCCP can provide customers with guarantees of VM protection in the cloud.

3.2 Concerns with TPM Usage in the Cloud Setting

TCCP highlights the role that cloud attestation can play in building trust in the cloud, namely by providing security guarantees that were not possible before. However, a closer look at the TCCP design brings to light some complications that are not immediately apparent. Unless these issues are properly handled, trusted cloud services built upon TPMs could incur performance, privacy, and management problems.

Performance issues. Today’s TPMs are not built for high performance, which is reflected in the poor latency and throughput of TPM primitives. Figure 3.2 presents the results of a performance benchmark of the core primitives of commodity TPMs, including quote, seal, and unseal. This experiment was conducted in several machines and performed by McCune et al. [MPP⁺08b]. As we can see, the execution time of a single quote operation takes between 300ms and 900ms. Benchmarks that we conducted using a Winbond v1.2 TPM corroborate this number, taking close to one second to complete. Since TPMs can execute only one single command at a time, the throughput of TPM

3 Towards Trusted Cloud Computing

primitives is low. The high latency and the small throughput of the TPM could negatively affect the scalability of cloud services that use TPMs. For example, in TCCP, the quote operation lies in the critical path of the attestation protocols invoked by the customers during VM creation and by the cloud nodes during migration operations. As a result, the number of operations per unit of time could be hindered due to the inefficiency of TPMs, creating bottlenecks. The low TPM performance could also open avenues for denial of service attacks because malicious customers could purposely keep the TPM busy and delay other customers' access.

Privacy issues. As we mentioned above, cloud providers are very keen on controlling the amount of information about the cloud internals that is leaked to the public. They tend to be hesitant to reveal about the number, characteristics, and placement of their machines, and about the internal configuration of the software systems. Their rationale includes concerns regarding competitors (preserving trade-secrets) and security (concealing potential vulnerabilities). In the TCCP architecture, by allowing customers to directly attest the cloud nodes, cloud providers are giving away the unique identities and the software hash values of every node. This information could be used, e.g., to infer the number of cloud nodes of the cloud infrastructure, and the distribution of different software platforms they run, and then leveraged for harming the cloud customers. In 2009, researchers managed to create a rough map of Amazon's cloud infrastructure and used it to place virtual machines on the same physical node [RTSS09]. They argued that this capability could enable an attacker to target a particular victim VM and try to learn secrets from it by exploiting side-channels. With cloud attestation implemented as in TCCP, external agents could gather more information and refine such maps. If revealing this information is unacceptable to cloud providers, alternative designs to TCCP need to be developed.

Management issues. The cloud is a massively distributed and dynamic environment, where, for fault tolerance and resource management reasons, the workload often migrates between clusters within the cloud infrastructure. In the TCCP architecture, this flexibility could be hindered. When resuming a suspended VM from the VM repository, it is likely that the cloud node for hosting the resumed VM is not the cloud node where the VM was suspended. In the interim, the workload conditions may have changed, thereby affecting the load balancing decisions leading to choosing a different node. It could also have happened that the original node was shut down to save power. In these circumstances, it is not possible to unseal the VM state, because the TPMs for sealing and unsealing would be different. Unsealing would also not be possible if the cloud node is preserved, but its software configuration changed, for example, due to a software upgrade. In both situations, the TPM unseal operation fails to return the decryption key for the VM state, thereby aborting the VM resume operation. The rigidity of sealed storage could then create hurdles to the management of the workload in the datacenters.

Although we use TCCP to illustrate these issues, they could also arise when building arbitrary trusted cloud services. At the root of these problems lies the fact that TPMs were not originally developed for the cloud environment. TPMs were targeted for single node platforms, namely desktops, and not for the distributed and dynamic environment where cloud services operate. Recent proposals for TPMs in the cloud do not address these limitations. Systems like Nexus [SdBR⁺11] or CloudVisor [ZCCZ11] use TPMs to allow users to remotely attest only a single cloud node and therefore do not address the preceding issues, but focus on the complementary problem of securing the platform running on a single node. While the TPM limitations could be addressed in the implementation of the trusted services themselves, the solutions would be hardcoded and not systematic. Therefore solutions will have to be repeated for every different service that needs to be protected, thus complicating the design and maintenance of these cloud services. To address this problem in a principled and general manner, we designed Excalibur, a system that offers the designers of trusted cloud services an abstraction—*policy-sealed data*—that enables them to take advantage of the TPMs’ properties while masking the limitations of the TPMs. The following chapter is devoted to presenting this system.

3.3 Summary

In this chapter, we presented the concept of a Trusted Cloud Computing Platform (TCCP). TCCP is a cloud architecture that provides an IaaS service for VM hosting. TCCP guarantees confidential execution of guest VMs and allows users to attest that the service is secure before they launch their VMs. TCCP serves primarily two goals. First, it illustrates the potential of cloud attestation and shows that this technique could be used for enhancing cloud services with security properties that were not present in their original design. Cloud attestation plays a fundamental and necessary role in providing such guarantees, and can be implemented using standard TPMs deployed on the cloud nodes. Second, the TCCP architecture helps clarify the challenges that TPMs could introduce in the design of trusted cloud services. To address these challenges, we designed a system called Excalibur, which we present in the next chapter.

4 Building Trusted Cloud Services with Excalibur

This chapter presents Excalibur [SRGS12], a system that overcomes the technical inadequacies of TPMs in cloud environments. Coupled with TPMs, Excalibur provides a key building block for designing trusted cloud services. Excalibur is presented in five sections: design goals, high-level design, detailed design, implementation, and evaluation.

4.1 Design Goals

The primary goal of Excalibur is to address the hurdles of using TPM primitives in designing trusted cloud services. As we explained in Chapter 3, TPM primitives, namely *quote*, *seal*, and *unseal*, could raise concerns regarding efficiency, privacy, and manageability if not used properly in the cloud. Excalibur aims to overcome these hurdles by providing a high-level programming abstraction (trusted cloud computing primitives) that the trusted cloud service developers can use instead of TPMs' low-level primitives.

Excalibur should meet the following design goals:

- **Simplicity:** Enhancing cloud services with cloud attestation should not incur a significant increase in complexity for either the developers and cloud administrators. Therefore, Excalibur's trusted cloud computing primitives should be simple and the burden of maintaining the system should be low.
- **Efficiency:** The system should not only mask the inefficiency of the TPMs located on the cloud nodes, but also not introduce bottlenecks that could hinder the performance and scalability of trusted cloud services.
- **Privacy control:** Excalibur should allow the cloud provider to control the degree of information that is revealed about the internals of the cloud. The semantics of the trusted cloud computing primitives should accommodate the ability for fine-tuned control of attestation information.
- **Management flexibility:** Excalibur's primitive must allow for securely storing data on an untrusted medium within the cloud without hindering the cloud provider's ability to migrate data and load balance.

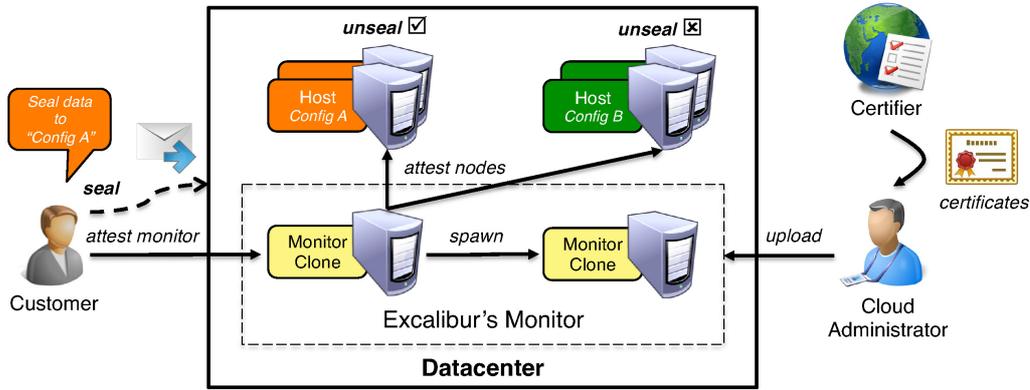


Figure 4.1: Excalibur deployment. The dashed lines show the flow of policy-sealed data, and the solid lines represent interactions between clients and the monitor. The monitor checks the configuration of cloud nodes. After a one-time monitor attestation step, clients can seal data. Data can be unsealed only on nodes that satisfy the policy.

4.2 Excalibur Design

We describe the design of Excalibur. After providing an overview of the system, we present each of the most important design aspects of the Excalibur in turn.

4.2.1 System Overview

Excalibur implements a trusted computing primitive called *policy-sealed data*, a simple programming abstraction for the developers of trusted cloud services. Policy-sealed data subsumes the functionality of TPM primitives without incurring risks of TPM misuse. It consists of only two operations: *seal* and *unseal*. Seal encrypts a piece of data and binds it to a customer-defined policy. Unseal is the only way to decrypt that piece of data; data can be decrypted on a cloud node if and only if the cloud node’s configuration satisfies the policy. The policy consists of a logic expression over a set of attributes, which refer to configuration features of a cloud node. In Section 6.3.4 below, we explain how policy-sealed data could be used in the design of trusted cloud services.

Excalibur exposes the policy-sealed data abstraction to the developers through a client-side library and enforces it in the cloud through a combination of cryptographic techniques and security protocols (namely the CPABE attribute-based encryption scheme) under the supervision of a centralized component named the *monitor*. Figure 4.1 illustrates a deployment of Excalibur, highlighting the separation between the *client-side library* and the *monitor*, which constitutes the heart of the system. The client-side library can be used on both the customer end (e.g., before uploading data) and by the hosting cloud nodes (e.g., before data migration).

The monitor is a dedicated service running on one or more cloud nodes—*monitor clones*. It coordinates the enforcement of policy-sealed data on the cloud. The monitor ensures that the policy-sealed data attributes are properly associated with the TPM state of each node, by mapping attributes (e.g., “vmm=Xen”) to the cloud nodes’ TPMs (e.g., “PCR=hash(Xen)”). Whenever a cloud node reboots, the monitor attests the booting machine and translates its TPM state to a set of attributes, which express the configuration of the cloud node. The monitor then encodes the attributes in special credentials that are sent to the node. These credentials are responsible for enforcing the unseal semantics: unsealing a policy-sealed data item fails unless the credentials of a cloud node are compatible with that item’s policy. In Excalibur, only the monitor can send requests that trigger TPM primitives on the cloud nodes, reducing the negative performance impact of TPM operations and preventing the exposure of infrastructure details.

Excalibur requires maintenance. For this purpose, the monitor exposes a narrow management interface to be used by the cloud administrator. The interface allows to configure the mappings between attributes and TPM identities as new software platforms and cloud nodes are deployed on the infrastructure. Configuration operations include adding and removing special *certificates* where these mappings are specified. Certificates are issued by trusted *certifiers* who vouch for the correctness of the mappings. In addition, the management interface enables the cloud administrator to scale up the system by spawning monitor clones. To bootstrap trust, customers can directly attest the monitor and thereby validate its correct operation and maintenance.

Next, we describe how the system works in more detail, starting with the policy-sealed data abstraction.

4.2.2 The Policy-Sealed Data Abstraction

The trusted cloud computing primitive provided by Excalibur is inspired by sealed storage, which we expand from the single-node setting to suit the needs of the multi-node cloud environment. Policy-sealed data allows user data to be bound to a cluster of cloud nodes whose configuration is specified by a user-defined policy.

Policy-sealed data offers two primitives for securing user data: *seal* and *unseal*. Seal can be invoked anywhere: either on the user’s computer or on the cloud nodes. It takes as input the user’s data and a policy and outputs ciphertext. The reverse operation, unseal, can be invoked only on the cloud nodes that need to decrypt the data. Unseal takes as input the sealed data and decrypts it *if and only if* the node’s configuration satisfies the policy specified upon seal; otherwise, decryption fails. Each cloud node has a configuration, which is a set of human-readable attributes. Attributes express features that refer to the node’s software (e.g., “vmm”, “version”) or hardware (e.g., “location”). A policy expresses a logical condition over the attributes supported by the provider (e.g., “vmm=Xen and location=US”). Table 4.1 shows an example of the attributes of a hypothetical deployment of a service akin to EC2. Table 4.2 illustrates the configuration of a particular node, and Table 4.3 lists example policies over node configurations in that deployment.

4 Building Trusted Cloud Services with Excalibur

Attribute	Value	Description
<i>service</i>	"EC2"	service name
<i>version</i>	"1"	version of the service
<i>vmm</i>	"Xen", "CloudVisor"	virtual machine monitor
<i>type</i>	"small", "large"	resources of a VM
<i>country</i>	"US", "DE"	country of deployment
<i>zone</i>	"Z1", "Z2", "Z3", "Z4"	availability zone

Table 4.1: Example of service attributes. In this case, EC2 supports two types of VM instances, two types of VMMs, and four availability zones (datacenters) in the US and Germany.

Node	Configuration
<i>N</i>	<i>service</i> : "EC2" ; <i>version</i> : "1" ; <i>type</i> : "small" ; <i>country</i> : "DE" ; <i>zone</i> : "Z2" ; <i>vmm</i> : "CloudVisor"

Table 4.2: Example of a node configuration. This configuration contains the values for the attributes that characterize the hardware and software of a specific node *N*.

Policy	Policy Specification
P_1	<i>service</i> = "EC2" and <i>vmm</i> = "CloudVisor" and <i>version</i> \geq "1" and <i>instance</i> = "large"
P_2	<i>service</i> = "EC2" and <i>vmm</i> = "CloudVisor" and (<i>zone</i> = "Z1" or <i>zone</i> = "Z3")
P_3	<i>service</i> = "EC2" and <i>vmm</i> = "CloudVisor" and <i>country</i> = "DE"

Table 4.3: Examples of policies. P_1 expresses version and VM instance type requirements, P_2 specifies a zone preference for one of two sites, and P_3 expresses a regional preference.

To secure user data on the cloud, policy-sealed data operations could replace the remote attestation and sealed storage assisted by TPMs. This substitution can be illustrated by looking at the TCCP protocols described in Section 3.1.3. To protect data upon upload, suspend, or migration, instead of using the TPM calls one could use seal the data to a policy containing the expression "*vmm=HardenedVMM*". If the destination cannot unseal the data, then its configuration does not match the policy; therefore, the node is not trusted from the perspective of the user who originally specified the policy. Naturally, the cloud provider must specify the attribute-value pair "*vmm*"-"*HardenedVMM*", and a certifier must issue certificates that vouch for the correct mapping between this attribute-value pair and the PCR values of the secure VMM software binary.

Policy-sealed data brings several benefits over the native TPM primitives. First, it gives the cloud provider additional control over the information that is leaked. Instead of being forced to always reveal the identities and software hashes of the cloud nodes, cloud providers have the freedom to define the attributes they deem acceptable. Second,

policy-sealed data allows for an improved management flexibility within the cloud. This is because any cloud node that satisfies the customer policy can unseal customers' data, and not just a single node. Lastly, policy-sealed data provides an additional bonus—a richer cloud attestation semantics. With our primitive, cloud providers can express attestation features that were not possible with TPM primitives alone, for example the location of the datacenters.

Policy-sealed data is enforced cryptographically, involving attribute based encryption and distributed protocols between the monitor and the cloud nodes, as we explain next.

4.2.3 Cryptographic Enforcement of Policies

It is challenging to cryptographically enforce policies in a scalable, fault tolerant and efficient manner. Since the mapping between the high-level abstractions (attributes and policies) and the low-level abstractions (TPM primitives) is done by the monitor, Excalibur must be carefully crafted so as to avoid bottlenecks in the monitor.

A first attempt to cryptographically enforce policies is to delegate this task to the monitor itself: upon sealing, the client encrypts the data with a symmetric key and sends this key and the policy to the monitor; the monitor then encrypts this key and the policy with a secret key and returns the outcome to the client. To unseal, the encrypted key is sent to the monitor, which internally recovers the original symmetric key and policy, evaluates the policy, and releases the symmetric key if the node satisfies the policy. Although this solution implements the necessary functionality, it involves the monitor in every seal and unseal operation and thereby introduces a scalability bottleneck.

An alternative design is to evaluate the policies on the client side using public-key encryption. Each cloud node receives from the monitor a set of private keys that match its configuration; in this scheme, each key corresponds to an attribute-value pair of the configuration. Sealing is done by encrypting the data with the corresponding public keys according to the attributes defined in the policies. This solution avoids the bottlenecks of the first approach because all cryptographic operations take place on the client side, without involving the monitor. Its main shortcoming is complicated key management due to the number of key-pairs that nodes must be handled in order to reflect all the possible attribute combinations usable by policies.

The solution we chose uses a cryptographic scheme called Ciphertext Policy Attribute-Based Encryption (CPABE) [BSW07]. This scheme first generates a pair of keys: a public *encryption key* and a secret *master key*. Unlike traditional public key schemes, the encryption key allows a piece of data to be encrypted and bound to a *policy*. A policy is a logical expression that uses conjunction and disjunction operations over a set of terms. Each term tests a condition over an attribute, which can be a string or a number; both types support the equality operation, but the numeric type also supports inequalities (e.g., $a = x$ or $b > y$). CPABE can then create an arbitrary number of *decryption keys* from the same master key, each of which can embed a set of attributes specified at creation time. The encrypted data can be decrypted only by a decryption

key whose attributes satisfy the policy (e.g., keys embedding the attribute $a = x$ can decrypt a piece of data encrypted with the preceding example policy).

Excalibur uses CPABE to encode the runtime configurations of the cloud nodes into decryption keys. At setup time, the monitor generates a CPABE encryption and master key pair and secures the master key. Whenever it checks the identity and software fingerprint of a cloud node, the monitor sends the appropriate credentials to the node, which include a CPABE decryption key embedding the attributes that correspond to the configuration of the node; the decryption key is created from the master key and forwarded to all the nodes featuring the same configuration. Sealing is done by encrypting the data using the encryption key and a policy, and unsealing is done by decrypting the sealed data using the decryption key. Policies are expressed in the CPABE policy language, which can be used to specify the examples in Table 4.3 as well as more elaborate policies.

The security of the system then depends on the security of the CPABE keys. The monitor protects the master key by (i) ensuring that it cannot be released through the monitor’s management interface, and (ii) encrypting it before storing it on disk, as described in Section 4.3.4. Additionally, cloud platforms must protect decryption keys. A software platform must prevent leakage or corruption of key material through its management interface (e.g., by direct memory inspection of VM memory); it must also hold the key in volatile memory so that key material is destroyed upon reboot. Moreover, the software platform must force a reboot after changing TCB components that get measured during a trusted boot (e.g., subsequent to upgrading the hypervisor). These properties ensure that the CPABE decryption keys of cloud nodes remain consistent with their TPM fingerprints and therefore reflect current node configurations. Section 4.3.4 explains how the CPABE decryption keys and the TPM configurations are kept synchronized as the cloud nodes reboot.

The benefits of using CPABE are twofold. First, it lets the system scale independently of the workload since the seal and unseal primitives do not interact with the monitor (and run entirely on the client side). Second, it permits the creation of expressive policies directly supported by the CPABE policy specification language while only requiring two keys – the CPABE encryption and decryption keys – to be sent to the nodes.

The cost of CPABE is high when compared to traditional cryptographic schemes. Section 4.3 explains how this impact can be minimized. A second drawback of using CPABE is key revocation, which is typically difficult in identity- and attribute-based cryptosystems. To handle revocation of decryption keys, our current design requires that all sealed data whose original policy satisfies the attributes of the compromised keys be resealed. This operation can be done efficiently by re-encrypting only a symmetric key, rather than the data itself.

4.2.4 Securing the Monitor

Since the monitor is managed by the cloud administrator, the mismanagement threats that affect any cloud node could also affect the monitor. One threat consists of adding flawed attribute mappings to the monitor that could compromise the semantics of policies. A mapping would be flawed, for example, if the attribute “*loca-*

tion=DE” were mapped to the identity of a node located in the US, or if the attribute *“vmm=HardenedVMM”* were mapped to the fingerprint of a non-hardened hypervisor. Another threat includes tampering with the monitor’s software binaries before booting the monitor with the aim of subverting the resolution of attributes in the monitor.

We address this challenge in two steps. First, we must first prevent the monitor from accepting flawed attribute mappings. To provide this guarantee, the monitor accepts only attribute mappings that are vouched for by certificates (see Section 4.3.3 for more details). Certificates are issued by one or multiple *certifiers*, who validate the correctness of mappings. A certifier could, for example, check that the cloud node with a certain AIK public key is located in Germany, and that a certain software hash corresponds to a valid implementation of a secure VMM. The certifier’s role could be played by the provider itself, or by external trusted parties akin to Certification Authorities. It is up to the customers to decide who to trust.

The second step aims to give customers guarantees about the integrity of the monitor. Only then the customers can trust that the system performs correctly. To provide such a guarantee, customers can directly attest the monitor’s software when first using the system. The monitor attestation also conveys to the customers the identity of the certifier that the monitor is using for validating the certificates. Customers can then decide whether the certifier is deemed trustworthy, and be sure that the certificate-based protections and the security protocols implemented by the monitor are correct. By enabling external attestation of the monitor, we must overcome several scalability bottlenecks, as explained below.

4.2.5 Monitor Scalability and Fault Tolerance

To improve scalability and fault tolerance, Excalibur supports multiple monitor clones. The cloud administrator can elastically launch or terminate monitor clones according to the workload. To evenly distribute requests among the clones, standard load balancers could be used. Clones are designed so that they do not need to communicate with each other for serving requests from customers or cloud nodes; only some sensitive key material needs to be securely exchanged when a clone is spawned, for which we developed the security protocol described in Section 4.3.4. This design enables the number of monitor clones to scale linearly with the workload.

To further improve the scalability of Excalibur, we further eliminate critical bottlenecks within a monitor clone. In particular, we introduce two optimizations. The first improves the throughput of monitor attestations triggered by the customers. This improvement is necessary, because using a standard TPM attestation protocol would incur bottlenecks that could hamper the practicality of Excalibur. Due to TPM inefficiency, the maximum throughput of a monitor clone would be bound to one attestation per second, clearly insufficient. To address this problem, we enhance the attestation protocol with a technique based on Merkle trees. This technique enables the monitor to batch a large number of attestation requests into a single TPM quote (i.e., signature of PCR registers by the TPM’s AIK key), dramatically increasing the throughput of the monitor attestation protocol (see Section 4.3).

<i>attest-monitor</i> (<i>mon-addr</i>)	$\rightarrow (K^{\mathbb{E}}, M)$ or FAIL
<i>seal</i> ($K^{\mathbb{E}}, P, D$)	$\rightarrow E = \langle P, D \rangle K, \langle K \rangle K^{\mathbb{E}}$
<i>unseal</i> ($K^{\mathbb{E}}, K^{\mathbb{D}}, E$)	$\rightarrow (D, P)$ or FAIL

Table 4.4: Excalibur service interface.

A second optimization within a monitor clone improves the throughput of decryption key requests issued by the cloud nodes. The algorithm for generation of CPABE decryption keys is inefficient, which could slow down servicing keys to the cloud nodes if a new key were to be generated per request. Since many machines in the datacenter share the same configuration (e.g., machines that belong to the same cluster), the monitor clone can instead securely cache the decryption keys and send them to all the nodes with the same profile.

4.3 Detailed Design

In this section, we present a more detailed view of Excalibur’s design. First, we describe the interfaces offered by Excalibur for building cloud services and managing the system. Then, we introduce the policy language, the certificates, and the security protocols of Excalibur.

Notation. Throughout this section, we use the following notation for cryptographic protocols. For asymmetric cryptography, K and K^P denote private and public keys, respectively. For symmetric keys, we drop the superscript. For the cryptographic scheme Ciphertext-Policy Attribute Based Encryption (CPABE) [BSW07], notation $K^{\mathbb{M}}$, $K^{\mathbb{E}}$ and $K^{\mathbb{D}}$ denote CPABE master, encryption, and decryption keys, respectively. The notation $\langle x \rangle_K$ indicates data x encrypted with key K , and $\{y\}_K$ indicates data y signed with key K . We represent nonces as n ; nonces consist of unique numbers whose goal is to detect message replays in protocols. The session keys used in the protocols consist of symmetric keys. Nonces and session keys are randomly generated.

4.3.1 System Interfaces

Excalibur’s interface has two parts: a *service interface*, which supports the implementation of cloud services, and a *management interface*, which lets cloud administrators maintain the system.

The service interface exported by the client library supports three operations, summarized in Table 4.4. Before the data can be sealed on the customer side, *attest-monitor* must be invoked to check the monitor’s authenticity and integrity. It returns the encryption key $K^{\mathbb{E}}$ needed for sealing and a *manifest* M , which contains the certificates needed to validate the monitor’s identity and fingerprint (see Figure 4.2). The manifest is passed to the customer, who learns from it which attributes can be used in policies and identifies the provider and certifier identities needed to decide whether the service is

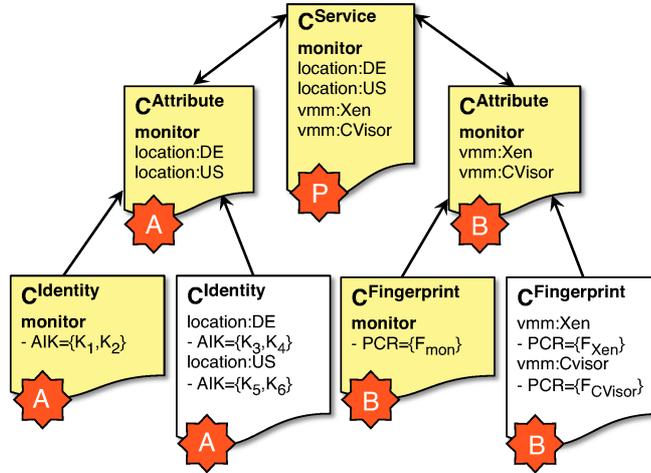


Figure 4.2: Example certificate tree and manifest. The certificates in light colored boxes form the *manifest* that validates the monitor’s authenticity and integrity.

trustworthy. Since the client saves both the manifest and the encryption key for future seal operations, *attest-monitor* needs to be performed only once when the cloud service is first used.

The core primitives are *seal* and *unseal*. Seal can be invoked by both cloud nodes and customers; it takes as arguments the encryption key K^E , a policy P , and the data D and produces an envelope E . This envelope is passed to *unseal*, which returns the decrypted data D or fails if its caller does not satisfy the policy. In addition to the decryption key K^D , *unseal* receives as an argument the encryption key K^E , which is required by CPABE decryption; the cloud node that invokes *unseal* must obtain this key from the monitor. *Unseal* also returns the original policy P so that a cloud node can re-seal the data with the customer’s policy. The CPABE policy language is used to express policies.

The management interface lets the cloud administrator remotely maintain the monitor using a console. Its main operations permit the administrator to initialize the system, manage certificates, and spawn monitor clones. Because these operations are intuitive and could be automated, we expect that the management cost of Excalibur will be relatively low for the cloud administrators.

4.3.2 Policy Specification

The policies that can be specified to create policy-sealed data blobs depend on two features: the policy specification language, and the set of attributes associated with a particular cloud service.

Excalibur adopts the policy language of CPABE [BSW07]. The features of this language enable the specification of expressive policies. The logic expressions support conjunction and disjunction operators, and can involve equality and greater / less than

operators between attribute values. Attribute values are typed, and two types are supported: strings and integers.

In addition to satisfying the rules of the policy language, all the attributes referred by a policy must be supported by the cloud service, otherwise the policy is ill-formed. The set of attributes associated with a particular cloud service must be specified by the cloud provider by publishing a *schema*. The schema indicates the names of each attribute and the domain of possible values that can be associated with each attribute. Upon sealing, all the attributes and values referred to in the user-defined policy are compared against the schema in order to guarantee that the policy is well-formed.

To define the schema of a cloud service, the cloud provider must take into account some restrictions. Namely, attributes can only refer to static properties related to the software or to the hardware of cloud nodes, i.e., attributes must refer to the PCR values and AIK public keys of the cloud nodes' TPM states. For example, “*VMM=Xen*” and “*Location=DE*” are valid definitions, because the former can be associated with PCR values matching Xen's software hashes, and the latter with the AIK public keys of nodes located in Germany. Attributes, however, cannot represent properties that change over time (since time cannot be specified in policies). As long as these constraints are followed, cloud providers are free to define the number of attributes, the names of attributes, and the possible values for an attribute. The meaning of attributes and values needs to be validated and declared in special certificates, whose format and usage we discuss next.

4.3.3 Excalibur Certificates

In this section, we describe the relevant issues related to certificates, namely their purpose, generation, format, validity, and management.

Purpose of Certificates

The primary purpose of certificates is to provide the monitor with a reliable method for representing cloud node configurations as a set of policy-sealed data attributes. Conceptually, a certificate is a statement signed by a certifier containing a mapping between attributes and the pair formed by the identity (AIK public key) and fingerprint (PCR values) of a cloud node. Whenever the monitor receives a quote from a certain cloud node in the process of node attestation, the monitor checks if the quote's AIK and PCR elements are covered by a certificate. If they are, then the configuration is deemed valid and the monitor proceeds with performing the translation into attributes and sending the corresponding CPABE credentials to the cloud node. Otherwise no CPABE credentials are sent because the configuration is unknown. Thus, so long as the certifier is trusted for properly mapping attributes to TPM primitives, the CPABE credentials sent to the cloud nodes match the cloud nodes' actual configurations and policy-sealed data is properly enforced.

Although the primary use of certificates is to validate the configuration of cloud nodes, we use them for another fundamental validation operation, namely for checking the configuration of the monitor. Note that there are two occasions where the monitor's con-

figuration needs to be validated. First, when customers attest the monitor they need to learn whether the quote they receive comes from a valid monitor clone, i.e., from a machine (i) deployed in the cloud provider’s premises and (ii) executing a trusted implementation of the monitor logic. Likewise, whenever a new monitor clone is instantiated, the candidate to play monitor clone must also satisfy safety conditions (i) and (ii). The question then is how can a monitor clone’s validator (i.e., a customer first using a monitor or an existing monitor clone spawning a new clone) check that the configuration of the monitor is valid.

To answer this question, we note that a similar problem has to be addressed by the monitor when verifying the configuration of cloud nodes, with the difference that we are now focusing on the configuration of the monitor itself. Since we already have a general mechanism for checking the configuration of any machine—the certificates—we leverage them to validate the configuration of the monitor. We designate a special attribute named “*monitor*” for mapping the AIK and PCR values that refer to trusted monitor configurations. Just like with any other configuration, there must exist a certificate covering the trusted monitor configuration. The validator of a monitor clone can then test its configuration like any other configuration by comparing a quote sent by the monitor clone against the “*monitor*” mapping contained in the monitor certificate.

In summary, certificates provide a unified mechanism for validating the configuration of a machine. Certificates are used on three occasions: (i) when the monitor attests the configuration of cloud nodes, (ii) when the monitor checks the configuration of a candidate monitor clone, and (iii) when a customer checks the configuration of the monitor. In (ii) and (iii), the attribute “*monitor*” is specifically validated.

Certification Procedure

Certificates are issued by one or multiple certifiers, who must vouch for the accuracy of attribute mappings. The methodology to validate such mappings depends on the specific attributes being certified. To validate a location mapping, for example, “*location=DE* → *AIKx*”, the certifier must check that the node with *AIKx* is placed in a cloud provider’s datacenter located in Germany. In addition, the certifier must control the management activity in the datacenters involving the deployment, displacement or decommissioning of cloud nodes so that the mapping holds over time. To validate the software mapping, the methodology is different. For example, for mapping “*VMM=Xen* → *PCRy*”, the certifier must confirm that the *PCRx* values correspond to Xen’s binary hashes. If the validation is correct, the certifier can then issue a certificate containing the respective mapping and signed with the certifier’s private PKI key.

Excalibur supports multiple certifiers, namely by allowing different certifiers to independently check specific attributes, and by allowing multiple certifiers to check the same attribute. There are several reasons for supporting more than one certifier. First, verifying all different attributes of a cloud service might require a number of skills that a single certifier may not gather. By supporting multiple certifiers, different entities may be hired to validate specific attributes for which they have expertise. Second, relying on a single certifier could raise privacy concerns in cloud providers because too many

internal details of the cloud infrastructure could be revealed to a single external organization. By supporting multiple certifiers, cloud providers can narrow down the amount of information that a single entity is allowed to obtain. Lastly, relying on a single certifier centralizes trust in a single entity. With multiple certifiers, trust is spread across multiple entities, thereby reducing risks.

Deciding on who to hire for certifying the cloud infrastructure is at the cloud provider's discretion. A certifier's role can be played by external organizations (e.g., Certificate Authorities), but also by dedicated departments of the cloud provider. Ideally, the set and identity of the certifiers should be chosen so as to increase customers' trust in the verification procedure. For this reason, hiring multiple, external, and reputable organizations may offer a preferable option.

The Format of Certificates

The question we now address is the format of the certificates. This issue is not as simple as one might first think due to the multiple requirements that must be taken into account. To show why, we start with a strawman format, clarify the requirements that need to be addressed, and present our solution.

First of all, certificates must address the basic needs of the system with respect to mapping attributes to identities and fingerprints. The simplest format for a certificate consists of a file containing the identities of the certifiers and a list of mappings $attr:value \rightarrow AIK/PCR$, and then have that file signed by all certifiers and by the provider. This format serves the needs of the two main parties that make use of the certificates: the monitor and the customers: (i) the monitor has all the information for checking the configuration of cloud nodes and of candidate monitor clones, and (ii) the customers can validate the identities of the certifiers responsible for validating the service by checking the provider's and the certifiers' signatures, and check the configuration of the monitor by checking the AIK and PCR values of the signature issue by the monitor's TPM against the "monitor" attribute.

This format, however, has two main drawbacks. First, it is very rigid. Even for minimal changes, e.g., adding or removing machines, all certifiers must agree to produce a new certificate to reflect the changes into the system. Second, this format might compromise the privacy of the cloud infrastructure. The certificate would disclose the AIK and PCR values of all the software and of the cloud nodes in the cloud infrastructure, and not just the details of the monitor (which are necessary for the sake of validating the monitor configuration). This side-effect would defeat the very purpose of policy-sealed data by revealing all this information.

These limitations prompted us to develop a more expressive certificate format. Excalibur certificates form a hierarchical tree like in the example shown in Figure 4.2. The example shows how a provider P can use the certificates that correspond to the internal nodes in the tree to delegate the certification of different attributes to two certifiers, A and B. Table 4.5 shows in detail the composition of each certificate type.

To allow for flexibility in producing the certificates, we split one single certificate into four different kinds of certificates, each of them containing a smaller number of state-

$C_{Provider}^{Service}$	$\rightarrow \{id_S, attribute[], K_{Certifier}^P\}K_{Provider}$
$C_{Certifier}^{Attribute}$	$\rightarrow \{id_A, attribute[], id_S\}K_{Certifier}$
$C_{Certifier}^{Identity}$	$\rightarrow \{id_A, attribute[], AIK[]\}K_{Certifier}$
$C_{Certifier}^{Fingerprint}$	$\rightarrow \{id_A, attribute[], PCR[]\}K_{Certifier}$
M	$\rightarrow C_{Provider}^{Service}, C_{Certifier}^{Attribute}[], C_{Certifier}^{Identity}(mon), C_{Certifier}^{Fingerprint}(mon)$

Table 4.5: Certificate and manifest formats. A certificate $C_{Provider}^{Service}$ identifies the service, the attributes, and the certifiers. A certificate $C_{Certifier}^{Attribute}$ identifies a list of attributes of a service vouched for by a certifier. Certificates $C_{Certifier}^{Identity}$ and $C_{Certifier}^{Fingerprint}$ validate identities and fingerprints, respectively. Manifest M comprises certificates of service, attributes, and monitor identity and fingerprint. Square brackets indicate a list.

ments that can be verified independently without requiring re-flushing and re-generating everything anew. There are certificates for four different statements: stating the identities of the certifiers responsible for each attribute (signed by the cloud provider), stating the attributes that a particular certifier is responsible for (signed by the respective certifier), and stating the mappings of attributes, which can be issued at different granularities (signed by the respective certifier).

In order to preserve the consistency of the strawman format, the various certificates must refer to other certificates, forming the hierarchical structure shown in Figure 4.2. This format addresses the flexibility requirement because certifiers are now able to easily and independently create certificates for the attributes they are responsible for without interfering with other certifiers nor being affected by other certifiers' activity.

In addition, the hierarchical structure overcomes the privacy concern relative to exposing the AIK and PCR values of the entire infrastructure. Instead of revealing all the information to the public (as the strawman does), only the subset of certificates that is necessary to validate the monitor's configuration is published, namely the leaf certificates covering the monitor's AIK and PCR values, and their respective parents all the way until the root (illustrated in a light shade in Figure 4.2). We call this subset the *manifest*. The manifest is sent to the customers during the monitor attestation protocol, enabling the verification of the signatures and monitor attributes by the customers without revealing detailed information about the cloud nodes (see Section 4.3.1).

Certificate Expiration

Excalibur includes a mechanism to limit the time period in which the certificates are valid (certificate expiration). This mechanism is relevant to the monitor, which uses certificates to check the configuration of cloud nodes, and to the customers, who use certificates to validate the monitor.

To support certificate expiration, every certificate includes an expiration date, which is set by the certifier upon issuing the certificate. The monitor must ensure that, once the expiration date has been reached, no more CPABE keys are issued for the attributes

covered by the expired certificate. To ensure that cloud nodes do not use obsolete CPABE keys, the monitor piggybacks the expiration data on the CPABE keys sent to the cloud nodes. As soon as the expiration time is reached, the cloud nodes drop the respective CPABE keys, and re-run the node attestation protocol to obtain fresh credentials. The monitor also checks the expiration date of the manifest before sending it to the customers, and aborts the protocol if the manifest has expired. Customers can also check whether the validity of the received manifest has expired.

Certificate Management

The certificates known to the monitor must be maintained by the cloud administrator, e.g., as a result of upgrading a software platform or deploying new cloud nodes. The monitor provides a specific management interface for adding and removing certificates.

To prevent trusting the cloud administrator for preserving the safety of the system, the monitor checks that the added certificates are consistent with the internal certificate tree maintained by the monitor. To be consistent, a certificate must satisfy the constraints implicitly defined by the signatures and statements contained in each certificate, as shown in Table 4.5. For example, when adding the certificate for location mapping “ $location=DE \rightarrow AIKx$ ” signed by certifier A , the monitor can only accept the certificate if A was endorsed by the cloud provider to vouch for this specific attribute. To enforce this condition, the monitor checks whether the certificate for the service contains statement “ $location \rightarrow A$ ”. If so, the monitor accepts the certificate, otherwise rejects it. This way, while the cloud administrator could compromise the availability of Excalibur (e.g., by switching down the monitor, or removing certificates), he could not tamper with the attribute mappings of the monitor.

The complexity of managing certificates depends on the number of certificates and the frequency of updates. The number of certificates can be greatly reduced thanks to the certificate tree structure. To avoid having a large number of certificates, the tree enables a single certificate to cover multiple attributes and multiple AIK and PCR values per attribute. The frequency of the certificate updates depends on how often the hardware and the software configurations change. The hardware configuration could change when cloud nodes are deployed or decommissioned. When compared to changes of the software configuration, it is less likely that the hardware configuration of deployed cloud nodes changes often over time. The configuration of software platforms could change as new software platforms go into production. This upgrade requires only uploading a certificate to the monitor with the PCRs of the new software.

Note that if a certificate is upgraded, causing the PCR values of an attribute to change, it is not necessary to re-seal the policy-sealed data that depends on the updated attributes. The reason is that, despite the changes in the attribute mappings, the monitor still sends CPABE decryption keys with the same set of attributes as before, allowing nodes to recover the policy-sealed data.

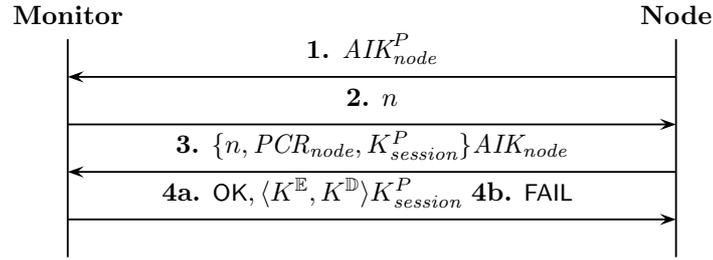


Figure 4.3: Node attestation protocol.

4.3.4 Excalibur Protocols

This section presents the security protocols of Excalibur: system initialization, node attestation, monitor attestation, seal and unseal, and clone attestation protocols. Excalibur’s protocols are fairly simple and elegant, which is achieved mainly due to the expressiveness of both (i) CPABE for cryptographically enforcing policy-sealed data, and (ii) Excalibur’s certificates for validating the configuration of a machine in a unified manner. We also highlight the incorporation of a new batching technique that enables scaling the throughput of the monitor attestation protocol.

System Initialization

Before the system can be used, the monitor must be initialized by binding a unique CPABE key pair to the service. To do this, the cloud administrator loads the certificates that validate the service attributes into the monitor and instructs the monitor to generate the key pair. If these certificates form a consistent certificate tree, the monitor creates unique encryption and master keys and binds them to the tree’s root certificate (see Figure 4.2). To permit system maintenance, the administrator can remove or add certificates as long as they form a valid certificate tree.

The monitor maintains its persistent state in a *certificate* store and a *key* store. Both stores keep their contents in XML files on a local disk. The certificate store contains the certificates loaded into the monitor. The key store contains all the CPABE keys. To secure the key material, the key store is sealed using the TPM seal primitive, which ensures that, in case the monitor reboots, the key store can be accessed only under a trusted monitor configuration.

Node Attestation Protocol

Once the setup is complete, the monitor delivers to each cloud node a credential that reflects that node’s boot time configuration, which will allow the node to unseal and re-seal data. The goal of the node attestation protocol is to deliver these credentials securely. Recall that, under our assumptions about the trusted software platforms running on the cloud nodes, when a cloud node reboots, the credentials kept by the node in

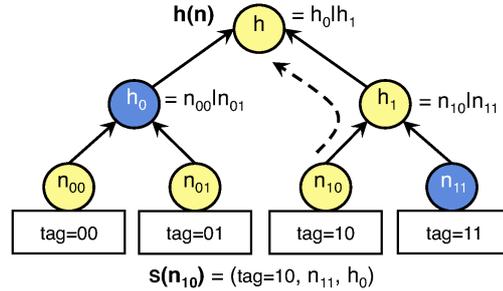


Figure 4.4: Batch attestation example. The tree is built from 4 nonces. A summary for nonce n_{10} comprises its tag and the hashes in the path to the root.

volatile memory are lost. Therefore, this protocol must be executed each time a cloud node reboots so it can obtain a fresh credential.

The monitor first obtains a quote from the node that is signed by the node’s AIK and contains the current PCRs. Then, the monitor looks in the certificate database for certificates that match the node’s PCRs and AIK. If any are found, the monitor obtains the node configuration by combining all the attributes of the matching certificates into a list like that shown in Table 4.2. Next, the monitor sends the credentials to the node; these include the encryption and decryption keys embedding these attributes. Since generating a new decryption key is expensive, the monitor caches these keys in the key store so they can be resent to nodes with the same configuration.

Figure 4.3 shows the precise messages exchanged between the monitor and a cloud node. The protocol is based on a standard remote attestation in which a nonce n is sent to the node (message 2), and the node replies with a quote (message 3); the nonce is used to check the freshness of the attestation request. Message 3 includes a session key $K_{session}^P$ that is used in message 4 to securely send credentials K^E and K^D to the node. Since the session key is ephemeral, an adversary could not perform a TOCTOU attack by rebooting the machine after finishing attestation (message 3) but before receiving the decryption key (message 4). Since the TPM only allows a 20-byte argument (the length of a SHA-1 hash) to be included in the quote operation, the monitor must first hash the two pieces of data n and $K_{session}^P$ (which exceed the maximum permitted length) in order to obtain the quote of message 3.

Note that the node does not need to authenticate the monitor to preserve the security of policy-sealed data. In the worst case, a node may receive a compromised decryption key from an attacker. However, given that customers seal their data with the encryption key obtained from the legitimate monitor, unseal would fail in such a scenario, and this attack would fail to compromise customer data.

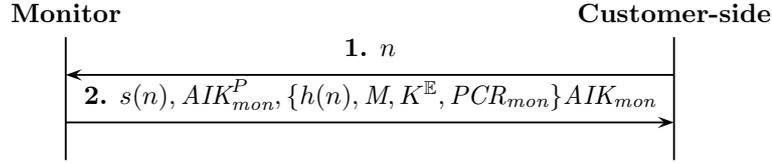


Figure 4.5: Monitor attestation protocol.

Monitor Attestation Protocol

The monitor attestation protocol is triggered by the *attest-monitor* operation, which lets customers detect if the monitor is legitimate by checking its authenticity and integrity. In addition, this protocol obtains: 1) the encryption key, which is used for sealing data, and 2) the set of certificates that form the manifest, which let the customer check the identity of certifiers and learn the attributes that are available. The monitor is legitimate if its identity and fingerprint are validated by the manifest.

The main challenge in designing this protocol is scalability. If every customer-side client were to run a standard remote attestation, then the throughput of the monitor would be extremely low due to TPM inefficiency.

To overcome this scalability problem, we batch multiple attestation requests into a single quote operation using a Merkle tree, as shown in Figure 4.4. The Merkle tree lets the monitor quote a batch of N nonces n_i expressed as an aggregate hash $h(n_{i=0}^N)$ and send evidence – summary $s(n_i)$ – to each customer-side client that its nonce n_i is included in the aggregate hash in a network-efficient manner (i.e., instead of sending all N nonces, it sends just a summary of size $O(\log(N))$).

The detailed monitor attestation protocol is shown in Figure 4.5. In the first message, the customer-side client sends nonce n for freshness and then uses the information returned in message 2 to validate the monitor in two steps. First, it checks in the manifest M for the certificates with attribute “monitor”; it uses them to authenticate the monitor key AIK_{mon}^P and to validate the fingerprint of the monitor’s software platform PCR_{mon} (see Figure 4.2). Second, to validate the freshness of the received messages, it compares nonce n and the summary $s(n)$ against the aggregate hash $h(n)$ produced by batch attestation. If all tests pass, the monitor is trustworthy, and the encryption key K^S is authentic. The customer can then seal data securely.

Seal and Unseal Protocols

The use of CPABE lets seal and unseal execute without contacting the monitor. In implementing these primitives, we take into account two aspects of CPABE related to performance and functionality. First, since CPABE is significantly more inefficient than symmetric encryption, seal encrypts the data with a randomly generated symmetric key and uses CPABE to encrypt the symmetric key. Second, given that CPABE decryption does not return the original policy (which unseal must return to let cloud nodes re-

seal the data), we include in the envelope the original policy and a digest for integrity protection (see Table 4.4).

Clone Attestation Protocol

To scale the monitor elastically, the cloud administrator can create multiple monitor clones. To do so, an existing monitor instance must share the CPABE master key with the new clone so the latter can generate and distribute decryption keys to the cloud nodes. However, this can be done only if the new clone can be trusted to secure the key and to comply with the specification of Excalibur protocols.

To enforce this condition, the existing monitor instance and the clone candidate run a clone attestation protocol analogous to that shown in Figure 4.3, but with two differences. First, after message 3, the monitor assesses if the candidate is trustworthy by checking whether its AIK and PCR values map to the “monitor” attribute contained in the manifest; if not, cloning is aborted. Second, if the test passes, the monitor authorizes cloning and sends the master key, the encryption key, and a digest to the candidate. The digest identifies the head of the certificate tree associated with the keys. The new clone refrains from using the keys until the administrator uploads the corresponding certificates to it.

4.4 Implementation

We implemented Excalibur in about 22,000 lines of C. This includes the monitor, a client-side library providing the service interface, a client-side daemon for securing the CPABE decryption key on the cloud nodes, a management console, and a certificate toolkit for issuing certificates. The console communicates with the monitor over SSL, and all other protocols use UDP messages. We used the OpenSSL crypto library [Ope] and the CPABE toolkit [acs] for all cryptographic operations, and we used the Trousers software stack and its related tools [Tro] to interact with TPMs.

We extended a cloud service so it uses Excalibur to help us understand the effort needed to adapt services for Excalibur and to estimate the performance impact of Excalibur on cloud services.

The example cloud service we adapted is an elastic VM service where customer VMs can be deployed in compute clusters in multiple locations, similar to Amazon’s EC2 service. Our extension used Excalibur to better assure customers that their VMs would not be accidentally or intentionally moved outside of a cluster in a certain area (e.g., the EU).

Our base platform was Eucalyptus [NWG⁺], an open source system that provides an elastic VM service with an EC2-compatible interface. Eucalyptus supports various VMs; we used Xen [BDF⁺03] because it is open source.

Our implementation modified Xen to invoke seal and unseal when the customer’s VM was created on a new node, migrated from one node to another, or suspended on one node and resumed on another. An attempt to migrate the VM to a node outside the

```

1324 sock.send("receive\n")
1325 sock.recv(80)
1326
1327 pipe = subprocess.Popen("/xen-/bin/seal",
1328     stdin=subprocess.PIPE,
1329     stdout=sock.fileno())
1330 fd_pipe = pipe.stdin.fileno()
1331
1332 XendCheckpoint.save(fd_pipe, dominfo, True,
1333     live, dst)
1334 os.close(fd_pipe)
1335 sock.close()

```

Figure 4.6: Hook to intercept migration (from file *XendDomain.py*.) We redirect the state of the VM through a process that seals the data before it proceeds to the destination on socket *sock* (lines 1327-1330).

specified locations would fail because the node would lack the credentials to unseal the policy-sealed VM.

Implementing these changes was straightforward. Integration with Excalibur required modifications to Xen, in particular to a Xen daemon called *xend*, which manages guest VMs on the machine and communicates with the hypervisor through the OS kernel of Domain 0. In particular, the VM operations *create*, *save*, *restore*, and *migrate* sealed or unsealed the VM memory footprint whenever the VM was unloaded from or loaded to physical memory, respectively. To streamline this implementation, we took advantage of the fact that *xend* always transfers VM state between memory and the disk or the network in a uniform manner using file descriptors. Therefore, we located the relevant file descriptors and redirected their operations through an OS process that sealed or unsealed according to the transfer direction. Figure 4.6 shows a snippet of *xend* that illustrates this technique applied to migration. Overall, our code changes were minimal: we added/modified 52 lines of Python code to *xend*.

The other two changes we made included (i) hardening the software interfaces to prevent the system administrator from invoking any VM operations other than the four noted above, and (ii) using a TPM-aware bootloader [gru] to measure software integrity and to extend a TPM register with the modified Xen configuration fingerprint.

4.5 Evaluation

This section evaluates the correctness of Excalibur protocols using an automated tool. We also assess the performance of Excalibur and our example service.

4.5.1 Protocol Verification

We verified the correctness of our protocols using an automated theorem prover. We used a state-of-the-art tool, ProVerif [Bla01], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus).

To use the tool, we specified all protocols used by our system, which included all cryptographic operations (including CPABE operations), a simplified model of the TPM identity and fingerprint, the format of all certificate types in the system, the monitor protocols, and seal and unseal operations. In total, the specification contained approximately 250 lines of code in pi-calculus.

ProVerif proved the semantics of policy-sealed data in the presence of an attacker with unrestricted network access. The attacker could listen to messages, shuffle them, decompose them, and inject new messages into the network; this model covers, for example, eavesdropping, replay, and man-in-the-middle attacks. ProVerif proved that whenever a customer sealed data, the resulting envelope could be unsealed only by a node whose configuration matched the policy.

4.5.2 Performance Evaluation

To evaluate Excalibur’s performance, we first evaluated the monitor’s scalability by measuring its performance overhead as well as its throughput for its three main activities: generating CPABE decryption keys, delivering these keys to nodes, and serving monitor attestation requests. We then measured the performance overhead of seal and unseal on the client side.

Setup and Methodology

We used two different experimental setups. The first used a two-node testbed; one node acted as a monitor, and the other acted as a regular cloud node making requests to the monitor. The second setup was used to evaluate the monitor throughput for attesting cloud nodes and serving customer attestation requests. For attesting cloud nodes, we simulated 1,000 nodes by using one machine acting as the monitor and five machines acting as cloud nodes, all running parallel instances of the node attestation protocol. For monitor attestations, we used a single machine acting as customers running parallel instances of the monitor attestation protocol. This number of nodes was sufficient to exhaust monitor resources and ensure that there were no bottlenecks in the client nodes.

Both setups used Intel Xeon machines, each one equipped with 2.83GHz 8-core CPUs, 1.6GB of RAM, and TPM version 1.2 manufactured by Winbond. All machines ran Linux 2.6.29 and were connected to a 10Gbps network. We repeated each experiment ten times and report median results; the standard deviation was negligible.

Decryption Key Generation

The overhead of generating a CPABE decryption key depends on the number of attributes embedded in the key. We measured the time to generate a decryption key stemming from the same master key, in which we varied the number of attributes from one to 50. This range seemed reasonable to characterize a node configuration.

Figure 4.7 shows the results, which confirm two relevant findings of the original authors of CPABE. First, the overhead of generating keys grows linearly with the number of attributes present in the key. Second, generating CPABE keys is expensive, e.g., a key

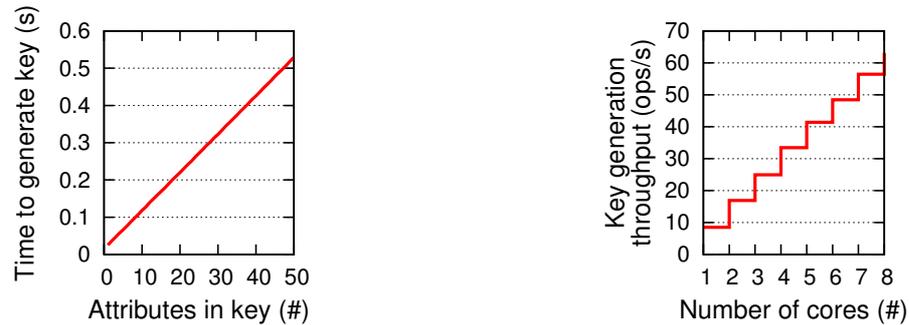


Figure 4.7: Performance of decryption key generation. Time to generate key as we vary the number of attributes (left), and throughput for 10 attributes as we vary the number of cores (right).

with ten attributes took 0.12 seconds to create, which corresponds to a maximum rate of 8.33 keys/sec on a single core.

Although CPABE key generation is inherently inefficient, we consider that its performance is acceptable since we expect the throughput pressure on the monitor to be relatively low because large groups of machines are likely to have the same configuration. The latency to generate a key is experienced only by the first node that reboots with a configuration new to the monitor. After the key is cached, it is reused in future identical requests without additional costs.

Node Attestation

The latency of the node attestation protocol was 0.82 seconds. The bulk of the attestation cost (96%) was due to the node’s TPM quote operation, which is necessary for remote attestation. This result is not surprising since such operations are known to be inefficient [MPP⁺08a].

Most of the work required by this protocol is carried out by cloud nodes. Therefore, the attestation latency should not represent a bottleneck to the coordinator. To confirm this, we evaluated the monitor’s throughput when running multiple parallel instances of this protocol. Results showed that the monitor could deliver up to 632.91 keys per second, which is efficient and would allow a single monitor machine to scale to serve a large number of nodes.

Monitor Attestation

We measured the performance of the monitor attestation protocol. This protocol had a latency of 1.21 seconds and a throughput of approx. 4800 reqs/sec on a single node. The quote operation performed by the monitor’s local TPM accounted for the bulk of the latency (0.82 seconds), and the remaining time was due to cryptographic operations and network latency. The high peak throughput we observed was enabled by batch



Figure 4.8: Performance overhead of sealing and unsealing data as a function of the complexity of the policy, with input data of constant size (1K bytes).

attestation. When we disabled batching, the throughput dropped sharply to 0.82 reqs/sec. Thus, this technique is crucial to the scalability of the monitor and delivered a throughput speedup of over 5000x.

Sealing and Unsealing

The performance overhead of the seal and unseal operations performed by Excalibur clients was dominated by the two cryptographic primitives: CPABE and symmetric cryptography (which uses AES with a 256-bit key size). We describe their effects in turn.

To understand the overall performance overhead of CPABE, we set the input data to a small, constant size. Figure 4.8 shows the performance overhead of sealing and unsealing 1KB of data as a function of policy complexity. On the left is the cost of a seal operation as a function of the number of tests contained in the policy. For instance, the policy $A=x$ and $(B=y \text{ or } B=z)$ contains three comparisons. Our findings show that the sealing cost grows linearly with the number of attributes. The cost of sealing for a policy with 10 attributes was about 128 milliseconds.

On the right, Figure 4.8 shows the cost of an unseal operation. Unlike encryption, CPABE decryption depends on the number of attributes in the decryption key that are used to satisfy the policy. For example, consider a decryption key with attributes $A:x$ and $B:y$, and policies $P_1 : A=x$, and $P_2 : A=x \text{ and } B=y$. Policy P_1 uses one attribute, whereas P_2 uses two. As before, the performance overhead of unseal grows linearly with the size of the policy. The time required to unseal a policy with 10 attributes was 51 milliseconds.

To study the relative effect of CPABE on the overall performance of Excalibur primitives, we varied the size of the input data. Figure 4.9 shows the fraction of overhead due to CPABE, and Table 4.6 lists the absolute operation times. Our findings show that CPABE accounts for the most significant fraction of performance overhead. Sealing 1 MB of data with a policy containing 10 leaf nodes took 134 milliseconds, and 87% of

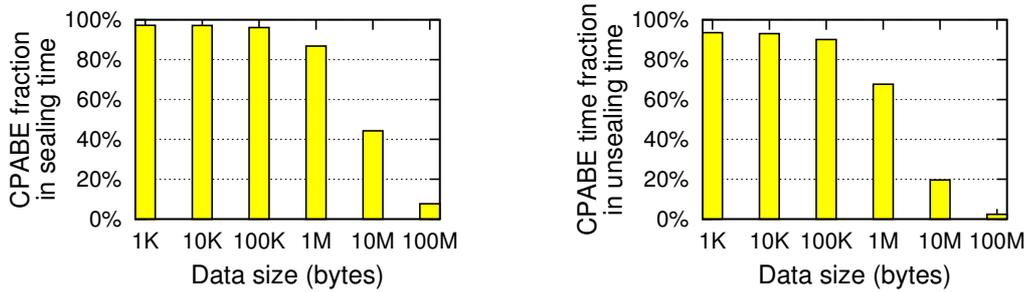


Figure 4.9: CPABE fraction in the performance overhead of sealing (left) and unsealing (right), varying the size of the input data.

Data (bytes)	Latency (ms)	
	Sealing	Unsealing
1K	120	50
10K	120	49
100K	121	51
1M	134	68
10M	264	243
100M	1522	1765

Table 4.6: Performance overhead of sealing and unsealing data, varying the size of the input data.

the total cost of sealing was due to CPABE encryption. For unsealing, the fraction of CPABE was slightly lower than for sealing, but it was still very significant. Unsealing 1 MB of data with a policy satisfying 10 attributes of the private key took 68 milliseconds, where 68% of the latency was due to CPABE.

In summary, our evaluation of Excalibur showed the following results: the costs of generating decryption keys and the node attestation protocol are reasonable when taking into account how infrequently they are required; the monitor scales well with the number of cloud customers that are using the service for the first time and with the number of cloud nodes that are attested upon reboot; the monitor could be further scaled up using cloning, and the latency of seal and unseal is reasonable and dominated by the cost of symmetric key encryption for large data items.

4.5.3 Cloud Compute Service

We now evaluate the performance overhead that the changes to Xen incur on its VM management operations, namely *create*, *save*, *restore* and *migrate*. We measured the time to complete each operation using an example VM for 10 trials. The example VM ran a Debian Lenny distribution with Linux-xen 2.6.26, used a 4GB disk image, and had a memory footprint of 128MB.

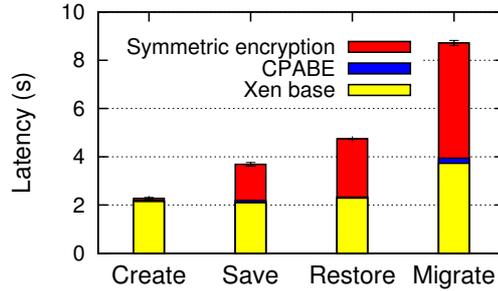


Figure 4.10: Latency of VM operations in Xen. Encrypting the VM state accounts for the largest fraction of the overhead, while the execution time of CPABE is relatively small. Encryption runs AES with 256-bit key size.

Figure 4.10 shows the results of our experiments. The performance impact is noticeable, especially for the *save*, *restore*, and *migrate* operations, where the completion time roughly doubled. The overhead, however, came from encrypting the VM’s entire memory footprint; using Excalibur to secure or recover the encryption key added a small delay. Unlike the other operations, *create* experienced a small overhead increase of only 4%. This is because the system only decrypted the kernel image, which occupied 4.6MB, instead of the larger VM footprint as it did for the other operations.

As the results show, seal and unseal introduced noticeable overhead to the VM operations due to the symmetric encryption of the VM image. However, given that these operations occur infrequently, and considering the additional benefits to data security, we argue that these results reflect an acceptable trade-off between security and performance.

4.6 Summary

This chapter presented Excalibur, a system that provides a new trusted computing primitive for the cloud setting. This primitive—*policy-sealed data*—enables the developers of trusted cloud services to take advantage of the properties of TPMs deployed on the cloud nodes for the purpose of building trust without facing the problems that TPMs could raise when used in the cloud environment. Policy-sealed data enables customers to specify a policy, containing the configurations they deem trusted for handling their data, and then seal the data before sending it to the cloud. Excalibur ensures that only the cloud nodes whose configuration satisfies the policy can unseal the data, hence recovering it. Excalibur provides a policy-sealed data service which is scalable, flexible, and easy to manage, thanks to the novel utilization of CPABE encryption and the development of a monitor component which can attest the cloud nodes very efficiently. We implemented Excalibur and evaluated it through benchmarks and integration with an open source cloud platform. The results showed that Excalibur is efficient and can

be applied to implement a trusted cloud service for the IaaS layer without burdening the developer with low-level TPM details. Excalibur can be seen as a building block for developing arbitrary trusted cloud services. We plan to pursue this line of research in the future, as we explain in Chapter 10. Next, we shift gears and focus on improving trust in enterprise environments.

Part II

Improving Trust in Enterprise Platforms

5 Motivation and Related Work

Many of the mismanagement risks discussed in Part I are not specific to cloud computing, but can occur more broadly within the scope of organizations in general. In fact, even if organizations outsource some of their data and processes to third-parties, they will still rely on in-house IT infrastructures—*enterprise platforms*—for handling some of their critical data. Given the nature of this data, instances of poor system administration could lead to security hazards and cause serious losses to organizations.

In this part of the thesis, we aim to strengthen trust in enterprise platforms by enhancing their security against IT mismanagement. To achieve this, our strategy is to enforce the principle of least privilege by limiting the privileges of administrators so as to reduce the window of vulnerability of the data. In particular, we propose *hierarchical administrator roles* so that most of the management tasks can be delegated to “untrusted administrators” without the fear of incurring violations to the confidentiality and integrity of users’ data and computations. In our scheme, only a small number of administrators needs to be fully trusted in an organization.

To enforce hierarchical administrator roles, the untrusted administrators need to be able to manage enterprise platforms without compromising security. To enable this, we make two contributions. First, we introduce and explore an untrusted-administrator operating system (OS) design named the *broker security model*. Our model provides guarantees of confidentiality and integrity of users’ data and computations against a malicious administrator, while retaining most of the manageability of the OS. Second, we demonstrate the viability of our model by building a set of extensions for Linux called BROKULOS. BROKULOS replaces Linux’s overly permissive management interface (i.e., superuser commands) with a narrow and carefully crafted set of commands (*brokers*) that enforce the security and manageability requirements of the broker model.

Before presenting our technical contributions in Chapter 6, we use the rest of this chapter to set the stage. We start by explaining in more detail the limitations of existing enterprise platforms and why these limitations undermine users’ trust. Then, to make these platforms more trustworthy, we propose the introduction of hierarchical administrator roles and discuss the challenges of enforcing them. Finally, we lay out our plan to address these challenges: we characterize our specific goals, assumptions, and threat model, and provide an overview of the related work.

5.1 The Problem of IT Mismanagement in Organizations

Organizations in general depend on the correctness of their IT infrastructures, and system administrators play a crucial role in keeping enterprise platforms operational. They

are responsible for a large number of maintenance tasks, most of which are security-sensitive, for example setting up access control policies, upgrading operating systems, and handling cryptographic keys. Because most of these tasks require superuser privileges, in practice, system administrators have full control over the OS and, consequently, over the user data hosted on those platforms. Given the privileged access that system administrators have to data, they must be trusted to manage the systems correctly and responsibly.

Unfortunately, security breaches have occurred in the past due to incidents of poor systems administration, resulting in loss, corruption, or leakage of data. In some cases mismanagement events were caused accidentally, prompted, for example, by the complexity of the systems or by simple negligence [Mas09]. In other cases, abuses of administration privileges were intentional, as illustrated by instances in which disgruntled employees have purposefully subverted systems of their organizations [CLM⁺, MCT], or by insiders that have stealthily misused data [goo10]. Surprisingly, some studies have recently shown that the risks of intentional data misuse are more prevalent than one should have thought. Surveys of employees across a variety of organizations showed that a considerable number of individuals would willingly steal secrets from their organizations, if they knew they were going to be fired [pol].

Prevention of security breaches due to IT mismanagement is not easy, especially in large organizations. First, in big organizations the IT infrastructure is larger than in small institutions, requiring more staff to maintain the systems. As a result, more people have privileged access to the data, making it more vulnerable to misuse. Second, in large organizations the web of relationships between employees is more complex and impersonal than in smaller ones. Consequently, it is more difficult to tightly scrutinize the behavior of individuals, and thus to detect and deter potential misbehavior.

In summary, irrespective of the cause, the risks of IT mismanagement incidents constitute a serious problem that could incur serious losses to organizations. The gravity of this problem has prompted us to find a solution for making enterprise platforms more robust to these threats.

5.2 Hierarchical Separation of Administrator Roles

Ideally, we would like the data security to be entirely independent of the administrators' behavior. In practice, however, it is hard to entirely eliminate the human factor from the equation. This is because certain management tasks require a great deal of control over the systems and privileged access to user data, e.g., to troubleshoot and recover from intricate failures, or to fine-tune the behavior of systems according to the needs of an organization; completely denying this degree of control would be too inflexible in practice.

Thus, rather than precluding trust in all system administrators, our strategy to improve security is to mimic smaller organizations by keeping the number of fully trusted administrators as small as possible as an organization grows. Specifically, we propose *hierarchical administration roles*, as represented in Figure 5.1. The idea is to create two

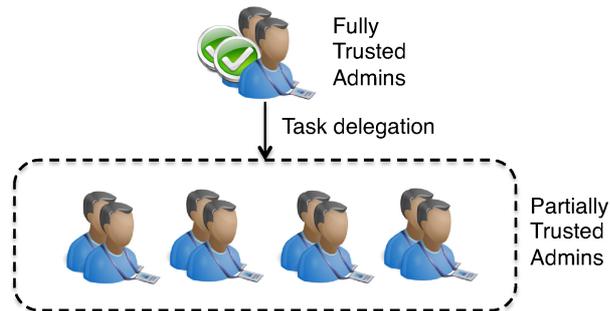


Figure 5.1: Hierarchical separation of administrator privileges in two roles: *fully trusted* and *partially trusted*.

administration roles with different privileges: *fully privileged* and *partially privileged*. While the former class retains full control over the systems, the latter class is trusted only for *resource availability*, but not for *information security*, i.e., to protect the *confidentiality* and *integrity* of data and computations. (By the lack of integrity we mean undetected modification of user data.) Thus, in contrast to a flat structure where all system administrators require superuser privileges to carry out their tasks, with hierarchical administrator roles a subset of these management tasks can be safely delegated to a class of “untrusted administrators”. This privilege separation contributes to keeping the number of fully trusted administrators small even in large organizations.

To be effective, this hierarchical separation of administration roles should allow for the delegation of a large number of management tasks to a partially trusted administrator. In principle, a management task can be performed by a partially trusted administrator so long as it can be performed without compromising the confidentiality and integrity of user data. However, this delegation is far from trivial in the current state of affairs. First, the OSes commonly used in enterprise platforms are not designed for supporting privilege separation between administrators. Most enterprise platforms run commodity OSes like Linux and Windows, which grant administrators superuser privileges. Second, most of the management tasks involve some form of access to user data, either direct (e.g., backing up data) or indirect (e.g., installing new applications), raising the question of whether a considerable fraction of such tasks could be completed without compromising the security of the data, and if so, what mechanisms would be necessary to implement this capability. Closing these technical gaps in the OS design space thus constitutes our main goal.

5.3 Goals, Assumptions, and Threat Model

In Part II of this thesis, our goal is to enable organizations to enforce the administrator privilege separation described above. We focus on administration roles targeting the OS and defer for future work the implementation of this privilege separation policy for other software systems that administrators also have to manage (e.g., databases). The OS must

provide for the security of users' data and computations while allowing administrators to perform the typical OS management tasks, such as installing applications, configuring devices, setting up security policies, and creating user accounts.

In devising OS defenses against mismanagement threats, we take two steps. First, we want to design a security model that can be applied to OSes in general. The security model should find a sweet spot in the design space that strikes a balance between limiting the power of the administrator and providing the functionality that is required for maintaining the system. We envision that the principles of our security model will be applicable to a range of software systems that currently depend on granting superuser privileges in their specific domains (e.g., database servers or web applications). Second, we aim to demonstrate the feasibility of our model when applied to a concrete commodity OS. Our solution should not require deep changes to existing OSes and should preserve compatibility with legacy applications. In Chapter 6, we present both parts of our solution, which include the *broker security model* and BROKULOS, respectively.

We design the broker security model and BROKULOS under the following assumptions. First, we assume that the OS administrator corresponds to the partially trusted administrator introduced in the previous section. The fully privileged administrators constitute the root of trust in the system, e.g., by vouching for the software that is trustworthy. Second, we assume that the implementation of the trusted computing base (TCB) of the system is correct. Our focus is not on minimizing the TCB size; such a goal is complementary to our work and has been the focus of various other research projects [MPP⁺08a, SK10, ZCCZ11, MLQ⁺10]. This allows us to focus on a popular operating system with a large TCB. Nevertheless, we discuss in Section 6.6 a possible approach to reducing the TCB size by using an information flow kernel such as HiStar [ZBWK06]. Third, we assume that the machine that hosts the system is physically secure, and that the system exposes a management interface that allows the administrator to manage the system remotely. This situation is common in many organizations that host and process sensitive data (see Chapter 2.3).

As for the threat model, we assume that the attacker can be impersonated by a rogue administrator, who has access to the management interface of the OS. In an insecure commodity OS, this interface includes all operations that can be executed with superuser privileges by logging into the root account or executing the *sudo* command. In the broker security model and in BROKULOS, the management interface exposes a much more restricted set of operations to the administrator. The attacker can also reboot the system and have access to the persistent system state stored on disk. The attacker, however, cannot exploit vulnerabilities in the TCB code of the OS, for instance, to perform privilege escalation attacks, nor perform physical attacks on the machine. In addition, we do not consider side channel attacks.

5.4 Related Work on Improving Trust in Enterprise Platforms

The related work on improving trust in enterprise platforms covers multiple topics. We organize these topics in four different approaches to increasing trust in enterprise plat-

forms: security models, isolation techniques, superuser privilege limitation, and Linux security mechanisms.

5.4.1 Security Models

Security models help us to reason about how the data can be accessed in an OS and who can access it. Bell-LaPadula [BLP76] and Biba [Bib77] are well known information flow security models for multilevel security. Just like other information flow control (IFC) models [ML97], they focus on how information flows in a system, and are dual to each other, allowing expressing confidentiality and integrity policies, respectively. These models, however, have not looked at reasoning about and expressing the permissions of the management operations required by administrators (e.g., for upgrading software), which is the focus of our work.

Another relevant security model is the Clark-Wilson (CW) model [CW87]. The CW is an informal security model specially designed for commercial purposes. It is concerned with data integrity, and it aims to prevent users from manipulating data objects arbitrarily. Users can only manipulate the objects through trusted programs, which streamline the way data objects can change (e.g., only certain users can perform certain transactions). Our broker model shares similarities with CW in that trusted programs also mediate certain activities in the OS. In contrast to CW, however, we focus not on users' access control but on administrators', and we go beyond CW in specifying concrete invariants that the trusted programs must adhere to in order to secure the administrator's management interface. (We elaborate on these invariants in Chapter 6.)

5.4.2 OS Isolation Techniques

In addition to security models, trust in enterprise platforms could also be reinforced by leveraging *isolation techniques*. In general, isolation techniques enable setting up security domains, some for the system administrators and others for users, such that the computations hosted in the users' security domains cannot be inspected or altered by the system administrators. The system administrators retain the control over the resources consumed by the users and can at any time release them, but cannot violate the confidentiality and integrity of users' runtime data. Since these properties are very much aligned with those we want to enforce (see Section 5.2), we review some representative techniques according to the isolation granularity that they implement (see Figure 5.2): *virtual machine*, *process*, and *function*.

Virtual Machine: The coarsest level of isolation granularity is the *virtual machine* (VM), in which a special hypervisor allows users to run VMs in security domains whose runtime state is isolated from the management domain. The administrator can control the resources of a guest VM but not access its data. (Note that commercial hypervisors do not offer such protections; the management domain gives the administrator full control over all the guest VMs in the system.) Terra [GPC⁺03] was the first hypervisor providing management isolation. Currently, the state-of-the-art includes CloudVi-

5 Motivation and Related Work

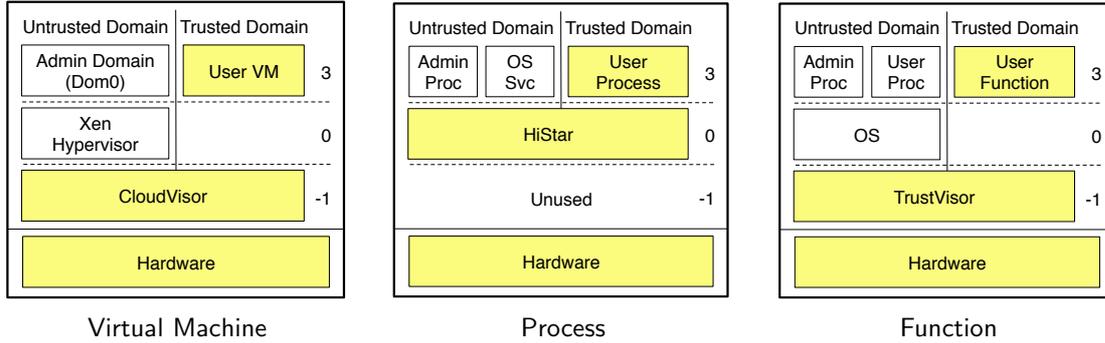


Figure 5.2: Representative systems that can isolate the runtime state of users’ computations from the administrator. Different systems enforce different isolation granularities: *virtual machine*, *process*, and *function*. The numbers in each diagram correspond to the protection rings found in Intel architectures, ranging from the most privileged (-1) to the least privileged (3) protection ring.

sor [ZCCZ11] and Credo [RRT⁺11]. CloudVisor, in particular, retrofits the management protections into Xen. To enforce such protections, CloudVisor leverages nested virtualization to run in a more privileged ring than Xen’s hypervisor and management domain (Dom0) (see Figure 5.2). In this way, CloudVisor allows the administrator to control the guest VMs’ lifecycle and associated resources (e.g., create, migrate, suspend, and terminate) without interfering with VMs’ states. While a VM-based solution is reasonable for virtualized platforms, adopting it for OS-based platforms has two drawbacks. First, it requires deploying an additional virtualization layer (e.g., CloudVisor and Xen) and transferring the native OS onto a guest VM. This additional layer introduces inefficiencies and additional complexity to the system. Second, since the administrator does not have any permissions on the OS now deployed in a guest VM, he would be precluded from managing the OS.

Process: Isolation can also be enforced at a *process* granularity. In HiStar [ZBWK06] and analogous systems [SdBR⁺11, HHF⁺05], the administrator can manage the CPU, memory, and bandwidth resources allocated to each process, but cannot access its internal state without the explicit authorization of the owner of the process. A microkernel enforces these protections by isolating the user processes from OS services and administrator processes. The biggest drawback of these systems is that they do not yet provide adequate support for system maintenance, mainly because this was not their main motivation. HiStar, for example, implements the influential Decentralized Information Flow Control (DIFC) model [ML97], but focuses primarily on the design of the microkernel, which is responsible for enforcing the DIFC policies. A real world OS, however, requires additional user-level processes with *declassification* privileges, processes that must appropriately handle user data during management activities. For instance, to offer data

backup services, some form of trusted daemon or declassifier must read the user data and forward it to the backup repository (perhaps after encryption). HiStar and similar projects did not look into the problem of securely designing the trusted daemons or declassifiers so as to cope with the range of management tasks performed by the administrators.

Function: Lastly, it is also possible to provide isolation of user computations at the *funcion* granularity. In Flicker [MPP⁺08a] and TrustVisor [MLQ⁺10], user processes can export security-sensitive functions to be executed in secure domains out of the system administrator's reach. Similarly to previous techniques, isolation is enforced by a tiny microkernel, which runs in a privileged protection ring. Applying these systems to securing enterprise platforms, however, raises manageability challenges similar to those of process granularity techniques. Furthermore, the programming models of Flicker and TrustVisor are likely to be too restrictive for securing most real world applications. Applications would need to be refactored in order to separate the security-sensitive functions from the security-insensitive parts of the application. In other words, existing applications would need to be rewritten, placing a considerable burden on developers and restricting the scope of computations that could be secured. Such a requirement would likely hinder the adoption of these systems in a real world setting.

In summary, much of this body of work has focused on the low-level kernel mechanisms that enable the construction of untrusted-administrator systems with *very small* trusted computing bases (TCB), an approach that is believed to lower the likelihood of security flaws. However, little attention has been devoted to building untrusted-administrator systems that actually remain *administerable*. Furthermore, the virtual machine granularity approaches burden users with OS management issues and adds unnecessary inefficiencies, and process and function granularity approaches require deep changes to applications and hamper programmability. Thus, the question remains open: how can we provide isolation from the management domain on OSES while preserving manageability, efficiency, and compatibility?

5.4.3 Restriction of Administrator Privileges

Some OSES have improved the security of user data by restricting the privileges of the administrator. Plan9 [iP02] was the first OS without superuser. Plan9 is a distributed system that comprises multiple nodes, each of which is managed independently by a node's owner. Although there is no system-wide superuser in Plan9, the owner of each node can control not only the node resources, but also compromise the security of the user data located on the node. HiStar [ZBWKM06] showed that the separation between resource management and data management is possible using DIFC. However, HiStar only provides the DIFC foundations for data protection and does not consider the high-level manageability issues addressed in BROKULOS. Similarly, trusted computing systems [MPP⁺08a, SdBR⁺11] have focused on securing user data and computations from the administrator by using confinement [MPP⁺08a] and labeling [SdBR⁺11] techniques,

but without specific requirements for preserving manageability. In the hypervisor world, the work by Murray et al. [MMH08] and more recently CloudVisor [ZCCZ11] allow for management of VMs without administrator interference, but address different challenges than BROKULOS's, which targets OSES rather than virtualized platforms. Some Linux distributions also try to mitigate the effects of accidental abuse of the superuser privileges. Ubuntu [ubu], for example, does not have a root account that the administrator can log into directly. However, Ubuntu does not preclude the administrator from acquiring superuser privileges and performing arbitrary operations. Therefore, it provides no protections against rogue administrators.

5.4.4 Security Mechanisms of Commodity Operating Systems

To implement access control, most modern operating systems support some kind of Access Control Lists (ACLs). Essentially, an ACL consists of a list that specifies the access rights of a set of principals over an object. For example, in Windows [Kei00], a principal could be a process acting on behalf of a user or service, an object could be a file, a directory, or a registry key, and an access right could be the permission to perform a certain operation on an object (e.g., “modify a file”, “list the contents of a folder”, or “delete subfolders and files”). Windows ACLs are very flexible, as they allow for fine-grained access rights and for the implementation of both Discretionary Access Control (DAC) and Mandatory Access Control (MAC) policies. However, ACLs are oblivious to the properties enforced by each of the permitted operations. The Broker Security Model (BSM) complements ACLs in that it specifies a set of security invariants that these operations must implement in order to allow for the delegation of security sensitive management operations to partially trusted administrators without compromising the confidentiality and integrity of users' data.

Many mechanisms have been specifically designed to improve the security of Linux. A large body of mechanisms aims to confine untrusted code to some kind of sandboxing environment. Notable examples include `chroot`, Jails [hKW00], Linux containers [lxc], and UserFS [KZ10]. Other mechanisms such as SELinux [Age01] and AppArmor [App] provide MAC support for Linux. However, because these mechanisms mostly focus on restricting access to user objects, they would considerably hamper the overall manageability of the OS, a drawback that we aim to overcome in our work.

Some of these proposals share similarities with BROKULOS in terms of policy enforcement. In particular, SELinux also allows for defining policies based on specific programs, but it differs from BROKULOS in that SELinux policies are defined by the administrator, whereas BROKULOS's policies are defined by the users. Just like in BROKULOS, AppArmor allows for attaching policies to programs based on file paths. However, in AppArmor, if a program has no policy associated with it, then it is by default not confined. Thus, contrary to BROKULOS, it cannot protect users from accidentally executing malicious programs not covered by the policies.

In summary, despite the differences and similarities between BROKULOS and the state-of-the-art Linux security mechanisms, the key contribution of BROKULOS is not so much in proposing fundamentally new mechanisms, but in showing that it is possible

to enhance Linux according to the broker security model by orchestrating well known Linux mechanisms, with little impact on performance and manageability.

5.5 Summary

This chapter discussed some of the trust issues that exist in the context of organizations. In particular, we have seen that organizations could incur severe losses if their enterprise platforms are poorly managed. To address this problem, we propose re-thinking the distribution of administration privileges by adopting *hierarchical administration roles*: rather than depending on a large number of fully privileged administrators, we keep this number small by offloading most of the management tasks to partially trusted administrators; this second class of administrators should be able to perform most of the management tasks without compromising the confidentiality and integrity of users' data. We laid out our goal of designing an OS that enables partially trusted administrators to maintain the OS. In the next chapter, we present the *broker security model*, which prescribes the key principles for designing such an OS, and BROKULOS, an OS that enforces these principles.

6 Enhancing the OS Security against Mismanagement Threats

This chapter presents our main contributions for improving trust in enterprise platforms: the *broker security model*, and BROKULOS. BROKULOS is an implementation of the broker security model for Linux. In contrast to a typical Linux distribution where the administrator holds superuser privileges, BROKULOS enforces privilege separation between fully trusted and partially trusted administrators such that most of the management tasks of the system can be performed by partially trusted administrators without compromising user data security.

In the remainder of this chapter we first present the key principles of the broker security model. Then, we describe how we applied these principles to the Debian Linux distribution and built BROKULOS. Lastly, we present our evaluation of BROKULOS and discuss some of its security features.

6.1 Broker Security Model

The broker security model aims to enhance the security of an operating system by weakening the trust requirements relative to the system administrator while preserving the manageability of the OS. Since we envision that the principles of our security model can be applicable to a class of software systems broader than OSes, namely those that grant some form of superuser privileges in their specific domains (e.g., database servers or web applications), our model includes a quite abstract and simple design, and proposes a methodology that system designers can follow to implement this design in concrete systems. We present this design and methodology below.

6.1.1 General Design

Figure 6.1 shows how the broker security model extends a base software system. The underlying system is modeled as a collection of objects, each of which is associated with a set of hardware resources and contains relevant data. If the base system is an OS, for example, objects include files, processes, user accounts, etc. The base system allows users and administrator to access and manage objects through two interfaces—a user interface and a management interface. In the base system the management interface gives the administrator superuser privileges, which allow him to fully control all system objects and therefore access user data without restrictions.

The broker security model introduces two main differences with respect to the base system. First, it implements hierarchical administrator roles, where instead of a single

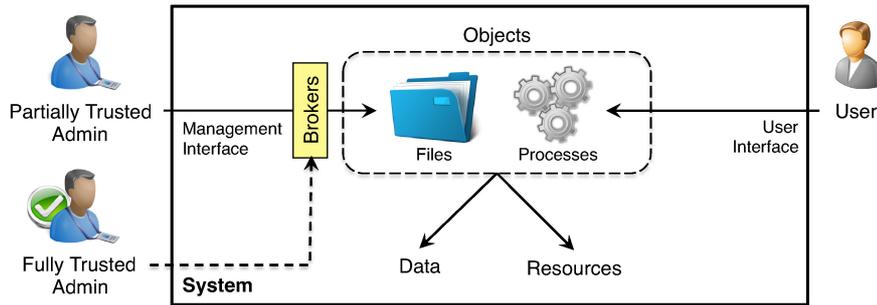


Figure 6.1: Software system under the broker security model.

all-powerful administrator, there are two administrator roles (see Section 5.2): *partially trusted*, and *fully trusted* administrators. Second, the management interface no longer grants indiscriminated superuser privileges, but provides administration privileges through a set of trusted programs called *brokers*. These programs are used by the partially trusted administrators to manage the system. Brokers mediate the access to objects in a well-formed manner in order to (i) provide the functionality that is necessary and sufficient to manage objects properly (e.g., create user accounts), and (ii) let the administrator retain control over resource availability while shifting control over user data confidentiality and integrity to users. The fully trusted administrators have the power to override the brokers' restrictions, for example, for troubleshooting.

To make sure that users retain control over their data security, brokers must be designed to preserve the following three security invariants:

1. **Information security:** A broker does not allow user data to be output or modified in ways that violate the confidentiality and integrity of that data. For example, allowing a debugger to be attached to a user process without the user having authorized or being aware of this operation violates this property.
2. **Identity protection:** A broker does not allow user identities and associated credentials to be hijacked or overridden. Otherwise, the administrator could abuse this privilege to impersonate a user and access his data. For example, allowing the administrator to change user passwords arbitrarily breaks this requirement.
3. **System integrity:** A broker ensures that the system can only transition between system states that preserve security invariants 1 and 2. For example, a broker cannot allow arbitrary kernel modules to be loaded because this feature could be exploited for privilege escalation: loading a malicious module could subvert brokers' security mechanisms.

The broker model has a direct correspondence to the hierarchical administrator roles introduced in Section 5.2. The partially privileged administrators are in charge of maintaining the OS and have access to the management interface, i.e., they can execute

the broker commands. The fully privileged administrators interfere more seldom in the maintenance process. They are primarily responsible for vouching for the broker implementations that correctly implement the security invariants and for overriding the brokers' protections in emergency situations (e.g., intricate system failures).

6.1.2 Methodology

Now that we have defined the broker model in abstract terms, we need to discuss how it can be applied to enhance the security of concrete software systems (and OSes in particular). For this, we propose a two-step methodology:

1. **Specify the broker functionality.** First, one must specify the broker layer by identifying the *functionality* that the set of brokers need to offer while simultaneously obeying the three security invariants required by the model.
2. **Implement the brokers.** Second, one needs to devise the *mechanisms* that implement brokers' functionality and enforce the security invariants.

We next apply both these steps to an OS. In the following sections, we refer to the partially trusted administrator role simply as “administrator”. Any reference to the fully trusted administrator role will be made explicit.

6.2 OS Broker Functionality

To adapt an OS to the broker security model, a natural design is to start by disabling superuser privileges and force the administrator to maintain the system from a regular user account. We can then grant that user account the privileges to execute a set of privileged commands through the `sudo` program. These commands will constitute the brokers that the administrator needs access to in order to maintain the OS. By this approach, we start from a point where the OS is secure by design, yet overly restrictive, and then add carefully crafted brokers to regain manageability.

The challenge then becomes specifying the functionality of brokers. In particular, we must make sure (1) not to overlook the functionality that is necessary for keeping the system administrable and yet (2) enforce the security invariants of the broker model. To properly specify the brokers' functionality, we start by surveying the most fundamental management tasks performed by administrators. The tools that support these tasks can provide us with the baseline mechanisms that we need to implement the brokers. However, since existing tools are likely to violate the invariants of the broker model, we need to validate whether and how such violations take place and complement these tools so that the resulting brokers can securely satisfy all invariants.

Table 6.1 shows the list of tasks that we surveyed along with an indication of how the various tasks violate the three security invariants we listed previously. This list combines the results of two approaches. In a bottom-up approach, we studied a collection of packages and respective tools available in a basic Debian distribution, identified the

6 Enhancing the OS Security against Mismanagement Threats

Category	Management task	IS	IP	SI
Software	List, install, upgrade, and remove applications and libraries executed by the users			–
	List, install, upgrade, and remove system services and kernel images			–
	Configure software and diagnose errors			–
	Apply security patches			–
	Manage local system documentation			–
Accounts	Create, modify, and delete user accounts		–	
	Disable user accounts temporarily			
	Modify account credentials		–	
	Force users to modify their credentials			
Groups	Create, modify and delete user groups		–	
Processes	Monitor and limit memory utilization by user processes	–		
	Check for runaway processes	–		
	Modify process execution priorities	–		
	Check for unattended login sessions	–		
Files	Perform backup and restore of user data	–		
	Set and view disk quotas			
	Check file space utilization	–		
	Remove temporary files (in /tmp and in /lost+found)	–		
	Re-distribute disk space in the filesystem	–		
	Mount and unmount filesystems			
	Check filesystem integrity and fight fragmentation	–		
	Check disk space	–		
	Create, modify, and format partitions			
System	Restart the system after panics, crashes, and power failures			
	Load, list, and unload kernel modules			–
	Start and stop services			–
	Automate and schedule system administration tasks with <code>cron</code>			
	Check and clear system log files	–		
	Configure and modify swap space			
	Configure <code>init</code> and runlevels			–
	Configure the network and check open connections	–		
	Setup system clock			–
	Setup and check the status of the printer			

Table 6.1: Management tasks grouped into categories: Tasks are grouped by category. For each task we indicate the security invariants they violate: information security (IS), identity protection (IP), and system integrity (SI).

functionality of each tool, and used our judgment to assess whether its functionality is fundamental for the administrator. In a top-down approach, we studied the system administration literature and identified the high level tasks that an administrator needs

to perform. Overall, we manually inspected 902 executables included in 100 packages¹ and studied three different textbooks [WOSW04, GBd04, Jos07]. We then converged on a single (coarse-grained) task list, which we have examined together with professional system administrators (from the host institution of the author) to make sure it reasonably characterizes the management activity of a typical OS administrator.

The tasks that violate the information security (IS) invariant mostly involve processes, files, and volumes and their primary goal is to manage resources and user data. For example, to learn about the memory utilization and open files by user processes, tools like `ps` and `lssof` reveal sensitive information that may be contained, e.g., in command line arguments of the process or in the names of user files. Similarly, tools for backing up and restoring user data (e.g., `tar` and `gzip`) allow the administrator to inspect and modify user data.

The tasks that breach the identity protection (IP) invariant are mostly related to user accounts and group management. User account operations include the ability to arbitrarily set and modify the identity and credentials of a user account (e.g., changing the password of an account using `passwd`). Group management enables adding and removing users from groups with tools like `useradd` and `usermod`. These capabilities would allow the administrator to access files and processes owned by the user, in the first case, or shared within a group, in the second case.

The tasks that compromise the system integrity (SI) invariant are mostly related to software and system management. Typical OSes allow the administrator to install arbitrary software, which can affect both the TCB (e.g., by upgrading the kernel, installing OS services, loading kernel modules) as well as shared applications. With this capability the administrator could escalate his privileges to access user data by tampering with the TCB or by installing backdoors in shared applications. Administrators can also set up devices to compromise the system integrity. For example, the ability to set the system time can be used to launch replay attacks.

Note that the purpose of Table 6.1 is not to enclose *all* management tasks. Instead, it comprises only the set of fundamental broker operations, which administrators can then rely upon for more complex tasks. For example, for diagnosing resource misuse, administrators can use various brokers, e.g., for checking runaway processes, unattended login sessions, and process memory utilization. In fact, it is typical to use helper tools to identify the source of such problems. As another example, for recovering from system bugs, administrators can use brokers for securely installing software and backing up or restoring user data. Indeed, rather than fixing compromised systems, the common practice for system recovery is to make clean-slate software reinstalls and restore user data from backups; this method guarantees that the system state is again known and trustworthy.

Ideally, the table should list all the tasks that are necessary and sufficient to meet all needs of OS administrators. However, in spite of our best efforts and positive feedback

¹These packages were selected from a minimal Debian distribution according to two criteria: they contain the basic tools (package “Priority” is “Required” or “Important”) and provide system administration support (package “Section” is “Admin”).

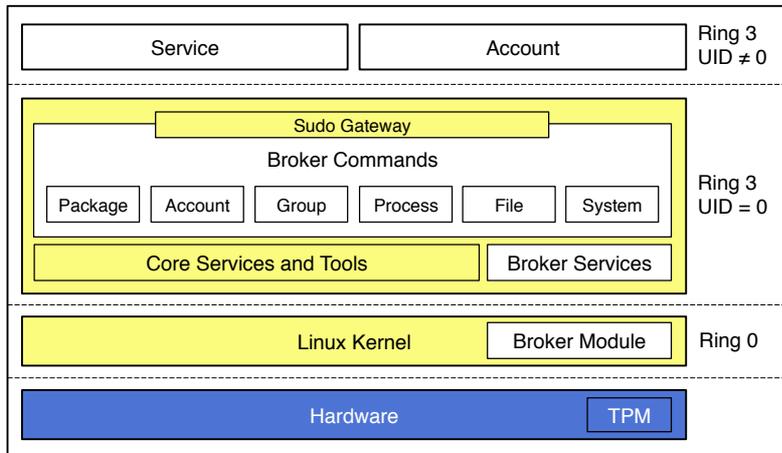


Figure 6.2: Broker-enhanced OS architecture. The numbers in each layer correspond to the traditional protection rings, ranging from the most privileged (0) to the least privileged (3).

from professional system administrators, this table is likely to be incomplete, and it may need to be adapted by adding, modifying, or removing entries depending on the concrete OS, deployment environment, and administrator needs.

Now that we have characterized the functionality that should be offered by the broker layer of an OS, we present the mechanisms that implement it.

6.3 Broker-enhanced OS Design

We start with an overview of the OS architecture that we propose and then describe how each security invariant is enforced by the brokers.

6.3.1 Architecture

Figure 6.2 illustrates the internals of a broker-enhanced OS. Since it is not our primary goal to minimize the size of the TCB, we simply extend a vanilla Debian Linux distribution with a set of components that implement the broker extensions for the system. These components consist of broker commands, dedicated services, and an LSM kernel module.

In contrast to the vanilla Debian distribution, there is no superuser account (root) nor any other way that the administrator can obtain superuser privileges. Instead, both users and the administrator run their processes in protection domains with $UID > 0$. $UID 0$ is then reserved for the components that need to run in privileged mode such as OS services (e.g., `init`, `sshd`) and broker commands. The space of unprivileged domains ($UID > 0$) is split into two parts: $UIDs \leq u_t$, which are reserved for services that do

not need to run in privileged mode, and UIDs $> u_t$, which are reserved for user accounts (where u_t is a configurable threshold).

Brokers consist of a well-defined set of trusted programs that run in privileged mode (UID = 0). Table 6.2 shows examples of the most representative brokers, grouped into categories according to their semantics. To allow for invoking brokers from a non-privileged account, we rely on the well known `sudo` gateway, which also authorizes broker execution based on the role—administrator or user—associated with each account. To bootstrap the creation of administrator accounts, the administrator role is assigned to the first account to be created; the administrator can then define the role of the subsequent user accounts.

Next, we describe in more detail the brokers that provide support for the management tasks in Table 6.1 while preserving the security invariants required by the model. We structure this presentation according to the invariants that are to be preserved.

6.3.2 Enforcing the Information Security Invariant

The information security invariant stipulates that the administrator cannot access user data through the system management interface. This is the model's most fundamental requirement because otherwise user data confidentiality and integrity could be directly violated. To meet this requirement, the protection domains of the administrator and users should be perfectly isolated from each other. However, this can be challenging when user domains must be crossed over, particularly for resource management and data management tasks. We discuss these in turn.

Managing Account Resources

The administrator must be able to control the resources associated with a user account (e.g., set user quotas for CPU and memory). This control, however, requires permission to access the resources allocated to user data. Without the proper protections, however, such access could allow the administrator to access user data, thereby compromising its confidentiality and integrity. To enforce a clean separation between resources and data, we propose the following steps.

The first step is to conservatively isolate the protection domains of administrator and users. To start, we can use the UID-based protection domains to prevent direct access to user files and processes that are not explicitly shared by the users. However, it is also necessary to prevent information leakage through the `/proc` filesystem. The Linux kernel exposes extensive information relative to user processes in a collection of files located under `/proc/PID`, where `PID` is the process number. The kernel generates the content of these files on the fly whenever they are opened and sets the permissions of many of them to publicly readable. However, making some of these files public violates the information security invariant (e.g., files `stat` and `cmdline` expose many details about the memory usage or the command line of processes, respectively). To prevent access to this information with minimal kernel changes, we simply override the file permissions to make them private to the process owner and accessible to the system brokers. We

6 Enhancing the OS Security against Mismanagement Threats

Category	Examples of representative brokers
Packages	list packages (<code>pkg-list</code>), get package (<code>pkg-get</code>), install package (<code>pkg-install</code>), upgrade package (<code>pkg-upgrade</code>), remove package (<code>pkg-remove</code>), flush package cache (<code>pkg-flush</code>)
Accounts	create account (<code>acc-create</code>), disable account (<code>acc-disable</code>), enable account (<code>acc-enable</code>), force password reset (<code>acc-force</code>), reset password (<code>acc-passwd</code>), delete account (<code>acc-delete</code>), load user policy (<code>acc-polload</code>)
Groups	create group (<code>grp-create</code>), list groups (<code>grp-list</code>), delete group (<code>grp-delete</code>), add member (<code>grp-addmem</code>), list members (<code>grp-lstmem</code>), remove member (<code>grp-remmem</code>)
Processes	list resource utilization (<code>ps-list</code>), kill account processes (<code>ps-kill</code>), set account process priority (<code>ps-renice</code>)
Files	backup account files (<code>fls-backup</code>), restore account files (<code>fls-restore</code>), list storage usage (<code>fls-du</code>), move account (<code>fls-move</code>), clean temp (<code>fls-cltmp</code>)
System	insert module (<code>mod-insert</code>), remove module (<code>mod-remove</code>), list services (<code>svc-list</code>), start service (<code>svc-start</code>), stop service (<code>svc-stop</code>), reboot (<code>sys-reboot</code>), setup system clock (<code>dev-clock</code>), setup network card (<code>dev-net</code>)

Table 6.2: List of representative brokers grouped into categories: States each broker’s functionality and command name (in parenthesis).

preserve kernel compatibility by adding these changes in an LSM module. Whenever a process issues the `open` system call to a sensitive `/proc` file, the LSM module checks if the UID of the running process matches the UID of the file (i.e., is its owner) and aborts the operation if not. To prevent a malicious administrator from bypassing these protections, the LSM module cannot be unloaded by the administrator.

The second step is to enable the administrator to manage account resources, and it consists of providing a set of specific brokers for process and file management. These brokers, however, only let the administrator “see” an account as a bundle of CPU, memory, and storage resources whose utilization he can observe, restrict (by setting quotas), and deallocate *as a whole*. For example, brokers for process management only output aggregate information of resource utilization and always operate on all processes of an account (e.g., by applying `kill` and `renice` to all processes). Brokers for file management follow the same approach. As additional examples, monitoring the storage consumed by a user only reveals aggregate disk utilization, and moving user files to another volume displaces all files located in users’ home directories or in user-approved subdirectories.

Exporting Account Data

The aforementioned techniques allow for resource management without user data access. However, in certain operations like backing up and restoring user data the administrator needs to export user data from the user account’s protection domain, where the data is secured, to another machine. To support these operations while preserving information security, the system encrypts the data and appends integrity checks before the data leaves the protection domain. However, we need to ensure that, when restoring the data, the backed up data can only be decrypted (1) on machines booting an untampered version of BROKULOS and (2) by the original owner of the data. To guarantee this

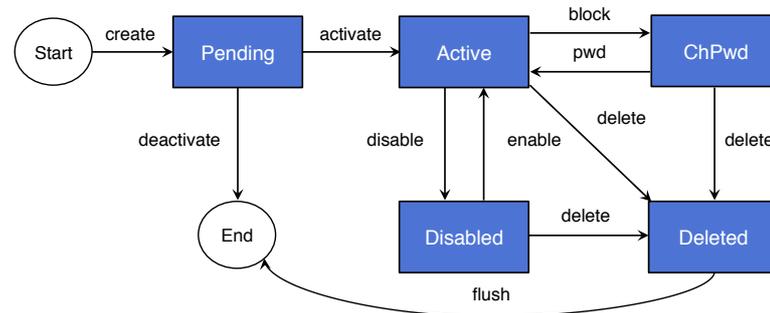


Figure 6.3: State transitions between account states: The user must explicitly accept that the account is valid before it can be used. In the active state, the administrator can temporarily disable the account or force the user to change authentication credentials. The resources of a deleted account can be released at a later point in time.

property, the user data is encrypted and decrypted with a *seal key*. The seal key is a unique cryptographic key that the system associates with each newly created account. To enforce requirement (1), we take advantage of TPM primitives, which allow us to encrypt (seal) the seal key such that it can only be decrypted (unsealed) if the machine boots a correct BROKULOS binary. If the booted system is correct, the system then ensures that the seal key is only accessible to the owner’s account, thereby ensuring requirement (2). To support recovering data on a different machine, e.g., because the original one was decommissioned, sealing could be extended to allow for unsealing to take place on any machine with a similar configuration. This extension could be implemented by coupling BROKULOS with Excalibur and sealing the data using the policy-sealed data primitive (see Chapter 4).

6.3.3 Enforcing the Identity Protection Invariant

With the protection mechanisms for the enforcement of information security in place, the administrator no longer has direct access to user data. Nevertheless, these protections could be circumvented if the identity protection invariant is not assured. This invariant requires that the administrator cannot control user credentials and identities, otherwise he could impersonate users and access their data directly. Thus, ideally, users should be able to control their own identities without hindering the administrator’s ability to control resources. In practice, however, shifting control to users entails some loss of management flexibility of accounts and groups from the perspective of the administrator. Therefore, we need to design specific brokers that can provide reasonable manageability without sacrificing the identity protection invariant, as we describe next.

Managing User Accounts

In managing user accounts, we enforce the identity protection invariant by offering a set of brokers for regulating an account's life cycle such that user login credentials are strictly controlled by the user.

The basic life cycle of a user account is shown in Figure 6.3. An account is created by the administrator; he specifies the initial configuration of the account (e.g., user name, home directory) and an initial login credential, which is only going to be used once. The first time the user logs in with the initial login credential, he must ensure that he has exclusive access to the account by claiming it. This process involves running a secure protocol which serves two purposes. First, it provides a report describing the initial account's configuration and state. If the account has been set up with initialization scripts or if somebody has logged into it before, the user will be able to detect these irregularities and abort the operation. If, however, the report shows no problems, the user can set up his authentication credentials (e.g., by uploading the user's public key) without administrator interference. This process will disable the initial login credential and lock the user name associated with the account. From this point onwards, only the user can login to his account and he has full control over its content, but not its resources. The administrator can still adjust the resources associated with the account, disable user login temporarily (e.g., in the case of a misbehaving user), force a user to change credentials, and, whenever necessary, delete the account.

Changing credentials is done by users themselves using the credentials they have uploaded to the system. To address the concern that losing user credentials would prevent a user from ever logging in, our system supports two override mechanisms. One is to rely on the fully trusted administrators to reset the user credentials. Another is to increase redundancy by registering multiple credentials and using various authentication mechanisms (e.g., public key, password, passphrase). Although this approach does not eliminate the problem entirely, it reduces the likelihood of permanent loss of access.

Managing Group Membership

In addition to allowing users to control their own identities and credentials, the members of user groups need to be properly authenticated. Otherwise, the administrator could gain access to group-shared data by creating fake identities and registering them as legitimate group members. To enforce the identity protection invariant when managing groups, the BROKULOS administrator is still allowed to create and delete user groups, but adding and removing members is delegated to users themselves. The approach we use for delegation is to designate a (per-group) *group leader* that makes group membership authorization decisions. The group leader must validate users' identities before adding them to a group. Since relying on user names chosen by the administrator is insecure for authentication, the group leader must check users' credentials (e.g., a certificate of the user's public key).

6.3.4 Enforcing the System Integrity Invariant

The mechanisms we have introduced thus far can effectively enforce both the information security and identity protection invariants. However, if the administrator can compromise these mechanisms, these assurances can no longer be guaranteed. Thus we next propose a mechanism for enforcing the system integrity invariant, taking into account two aspects of the problem: managing TCB components and shared applications.

Managing TCB Components

Managing TCB components involves installing, upgrading, configuring and removing software components that run in privileged domains and configuring devices (e.g., setting up the network and the system timer). The privileged software components include those in the kernel space (i.e., the kernel itself or kernel modules) and in the user space with UID 0 (e.g., services, system libraries, system tools, and brokers). To enforce the integrity of the TCB, all these operations must be validated, and this is carried out using special-purpose brokers.

In particular, brokers only authorize the installation of TCB components if the new TCB component is “trusted”. Several definitions of trust could be used, for example, in an ideal world, the system would automatically verify if the implementation is correct. BROKULOS uses a simple model where a TCB component is trusted if its compliance with the broker security model is endorsed by one or multiple third parties that are mutually trusted by both the fully trusted administrators and users, referred to as Mutually Trusted Signers or MTSes. To enforce this consent, administrators set up the initial MTS certificates in the system and users must approve or reject them whenever they claim their accounts. MTS certificates can be changed over time—e.g., when updating or revoking them, or when adding new MTSes—by either establishing a chain of trust that only accepts new MTS certificates signed by a preexisting MTS, or by polling all users before accepting a new MTS certificate. The MTS role can be performed by any entity mutually approved by administrator and users (e.g., certification organizations, software development companies, specific administration roles within the organization, or open source communities).

Regarding device configuration, we again only accept configurations that are vouched for by an MTS. The notion of what is expected from a trusted configuration is device-specific. Therefore device-specific brokers are expected to perform the appropriate validations. A particularly interesting case is the system clock, where the system time should not be set arbitrarily. Therefore, we restrict time updates to trusted NTP servers sent over secure channels. This is done by requiring the NTP configuration file (which identifies addresses and credentials of the NTP servers) to be signed by an MTS. Given the large number of devices, we did not design brokers for all of them, but new devices could easily be accommodated by adding appropriate brokers.

In addition to enforcing TCB integrity, it is necessary to assure users of its enforcement. This is because the administrator can circumvent the TCB protection mechanisms by rebooting the machine and tampering with the TCB binaries on disk. We offer these

guarantees by means of a remote attestation protocol, which users run when they claim their accounts. Our protocol is based on a standard attestation protocol [PMP10], which transmits the boot time measurements (hash) of the TCB components signed by the TPM. We then extend it to include the MTS identities as well as the report of the user account's initial state (see Section 6.3.3). Thus, when users claim their accounts they can validate the hashes of the TCB binaries and the MTS identities, thus verifying the integrity of the TCB.

Managing Shared Applications

Finally, in addition to TCB components, another type of software that must be trusted to correctly manipulate user data is shared applications (e.g., MySQL). To give users the flexibility of choosing which applications they trust, we let them define *user policies* that express their restrictions. The policy language expresses a list of rules, each of them consisting of comparisons among four attributes we currently support: package maintainer, package name, package version, and filename.

To enforce these policies, we developed a special purpose Linux Security Module (LSM) kernel module. The LSM module overrides the standard DAC permissions and enforces the user policy at runtime: whenever the user runs an external program, the LSM module intercepts this operation, evaluates the policy, and aborts the execution if the policy evaluation fails. To evaluate each policy rule, the LSM module checks the attribute conditions specified in the policy against a set of extended filesystem attributes associated with the executable. The filesystem attributes are attached by the broker layer whenever the executable's package is installed. The broker responsible for installing the packages obtains the attributes for each program from a manifest contained in the program's package. Users load their policies into the LSM module once they claim their accounts.

6.4 Implementation

Our BROKULOS prototype is based on the Debian GNU/Linux 6.0 ("Squeeze") distribution running Linux 2.6.39.3. Our implementation effort includes the broker layer, which we implemented in about 4,400 lines of Python code, and the LSM kernel model, coded in less than 1,000 lines of C code. For convenience, brokers take advantage of basic tools such as `dpkg`, `gpg`, and `useradd` to perform the low level changes to the system. These tools are included in the core packages of BROKULOS, which comprises 77 packages, out of a total of 266 packages. This package configuration is based on Debian's minimal setup, which is then extended with BROKULOS's functionality.

The LSM module implements the protection mechanisms for overriding the DAC permissions of the `/proc` files and evaluating user policies. To implement this functionality, it places handlers in two LSM hooks (`bprm_check_security` and `inode_permission`). The LSM module provides an interface via VFS under the mount point `/broku1os` for loading the user policies into the module.

Our current prototype uses TPMs to support remote attestation and secure storage. We use TrustedGRUB [`gru`] to measure the integrity of the files of core packages and

extend the PCR registers with these measurements accordingly. Then, we use the TPM's `quote` primitive to generate and sign an attestation report when requested by the users. This procedure requires setting up an AIK key so that the TPM can sign the report. The implementation of secure storage has some limitations: we keep the entire system on an encrypted partition using LVM, but, as of now, we have not modified LVM to ensure that the encryption keys are protected using the sealing primitives of the TPM. This modification, however, poses no particular challenges and is already used in Windows by BitLocker [Mic].

6.5 Evaluation

In this section we evaluate the security, manageability, and compatibility of BROKULOS, and experimentally gauge its performance overheads.

6.5.1 Security

BROKULOS improves security in three main ways. First, it significantly reduces the management interface exposed to the administrator. Unlike a commodity Linux distribution where the administrator is endowed with superuser privileges, in BROKULOS the administrator can only perform the privileged operations exposed through the broker layer. The broker layer makes the management interface explicit, and narrows it to a relatively small number of trusted programs. Thus, provided these programs are correctly implemented, the administrator cannot acquire privileges not contemplated in the broker model.

Second, BROKULOS explicitly restricts the software that can run in a privileged domain, i.e., that belongs to the TCB. In a commodity Linux distribution, because the administrator can install arbitrary software in the privileged protection domain, it is not possible to foresee which security properties are guaranteed by the system. In BROKULOS, however, only the software that is signed by an MTS can run in the privileged domain. Thus, provided that the MTSes are trustworthy, the system enforces the well-defined security invariants of the broker model.

Finally, BROKULOS allows users to specify the software they trust to process their data. BROKULOS conservatively prohibits the execution of all shared programs (i.e., not owned by the user) and allows the user to open exceptions based on a user policy. This mechanism prevents the user from accidentally running applications that could compromise the security of his data.

An orthogonal aspect of the system security is shrinking the TCB size to reduce the likelihood of code vulnerabilities. As we mentioned, this aspect was not the emphasis of our work and we therefore see it as being complementary and a follow up to BROKULOS. Nevertheless, we note that while brokers add code to the TCB, it is only a small additional fraction of much simpler code when compared to the OS kernel. Furthermore, we expect to make broker programs more trustable by releasing their source code.

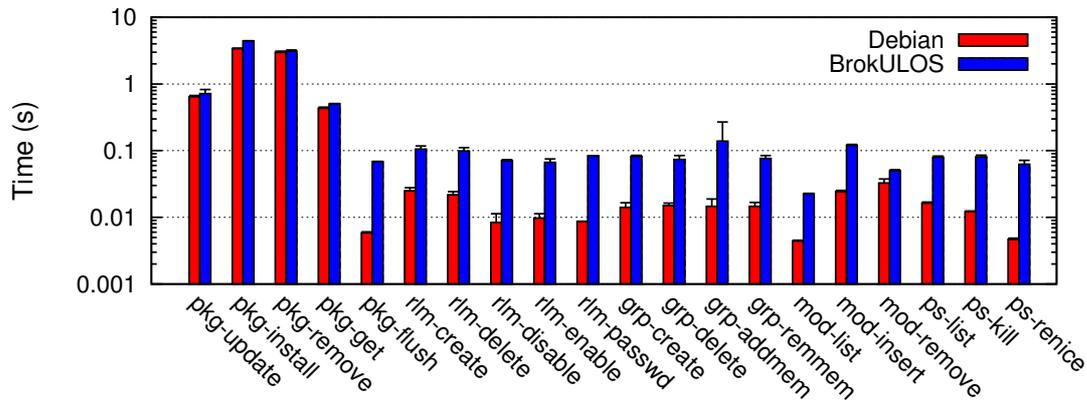


Figure 6.4: Performance of brokers when executed by the administrator: Covers representative brokers relative to package, account, group, module, and process management. The brokers for installing, getting, and removing packages use the `hello` package, which suffices for measuring the broker overhead for any package.

6.5.2 Manageability

The ideal way to evaluate the system manageability would be through the practical experience of deploying and managing the system in a real setting. Not having access to such a deployment, our methodology is to validate the whether BROKULOS provides adequate broker coverage to accommodate all the management tasks we have surveyed (see Table 6.1).

Our current prototype provides a set of 41 brokers spanning multiple task categories. In some cases there is a one-to-one correspondence between the task and a particular broker (e.g., backing up data is supported by `file-backup`), whereas in others a single broker serves multiple tasks (e.g., `ps-list` lists both the CPU and memory allocated to an account). Overall, BROKULOS currently covers the most crucial set of management tasks. We provide only limited support for tasks related to devices (e.g., managing the printer) and filesystems (e.g., formatting partitions and reducing fragmentation). Overall, out of the 33 coarse-grained tasks in the table, our system fully supports 29. The high fraction of management tasks covered by the existing brokers shows that our system provides extensive management support.

6.5.3 Compatibility

Overall, BROKULOS preserves compatibility with existing Linux mechanisms and applications. Our solution requires no modifications to the Linux kernel besides plugging in a kernel module to the standard LSM interface. The system leaves ABI / APIs unchanged, thereby preserving application compatibility. However, some popular administration tools are disabled, since they violate the broker model. This is the case,

for example, of `lsdf`, which prints out a list of every file that is in use in the system. As a result, the administrator may have to adapt and possibly change his scripts to use BROKULOS's brokers.

6.5.4 Performance

To evaluate the performance of our prototype, we focused on the places where BROKULOS introduces overheads to the vanilla Debian distribution: the broker layer, which affects management operations, and the LSM module, which impacts the execution latency of all programs in the system. (Recall that the LSM handler code runs every time a program is executed.)

Our evaluation methodology is as follows. To study the broker layer overhead we use microbenchmarks. For each broker, we measure its execution time, measure the execution time of a vanilla Debian operation whose functionality is comparable to the broker's (e.g., user account creation), and then compare both values to analyze the performance penalty incurred by BROKULOS's management tasks. For each experiment, we run 10 trials and report the mean time and standard deviation. To study the overhead of the LSM module we measure the impact of policy evaluation on the execution time of a large task, namely compiling the Linux kernel 2.6.39.3. We measure the overall execution time with and without policy evaluation, using a policy with 266 rules, where each of them tests a package installed in the system. We use an Intel Xeon machine with a 2.83GHz 8-core CPU, and 1.6GB of RAM.

Figure 6.4 plots the results of the broker layer evaluation. It shows only the subset of system brokers that (1) require sanitization of standard administrator tools to enforce compliance with the broker model (e.g., resetting the network card is not shown), and (2) have a direct correspondence with a vanilla Debian operation (e.g., the backup broker is not shown). There is a significant disparity in the performance overhead among brokers. Brokers whose Debian counterpart execute in the order of 10ms undergo a performance penalty of around one order of magnitude. For execution times above the 0.1s threshold, however, the performance penalty between the two cases is negligible. The high overhead of short-lived brokers is partly due to the extra functionality, but mostly due to being implemented in Python, whereas their Debian counterparts are implemented more efficiently in C. If we consider, e.g., the `ps-renice` broker, which sets the same priority for all the processes of a user, and its counterpart, which corresponds to the command `renice -u`, the 10-fold increase is simply due to Python overhead. Since the broker functionality is not significantly more complex than that of pre-existing tools', we believe that implementing brokers from scratch and in C should produce comparable performance to the Debian distribution.

Our LSM module study shows that policy evaluation is efficient. The overall execution times of the kernel compilation in Debian and in BROKULOS show no differences, which means that the LSM module adds negligible overhead to long running tasks. These results are expected since the LSM module handlers perform very little work and only when a program is executed.

6.6 Discussion

In this section we discuss several issues regarding possible design extensions and the deployment of the system.

Shrinking the TCB size. Several directions could be taken in re-designing BROKULOS's internals to reduce the TCB size. One direction is to leverage existing sandboxing mechanisms for Linux such as UserFS [KZ10] in order to run some of the trusted programs (e.g., privileged services) in an unprivileged environment. Thus, if one of these services is exploited, an attacker could not compromise the entire system. To avoid depending on the correctness of the large Linux kernel, a second direction is to explore designs based on microkernels [KEH⁺09] or on DIFC kernels [ZBWKM06, KYB⁺07]. The important thing to note is that the broker security model is also applicable in this setting, with the added advantage that brokers can set fine-grained policies. E.g., the `ps-list` broker can be constrained to only be able to read the `/proc` files. Thus, in the event of an exploit, the attacker could only leak the information contained in those files and nothing else, which significantly improves security.

Integration with distributed systems. In real enterprise platforms, hardly a single machine operates autonomously; machines usually rely on networked services for storing data (e.g., NFS), authentication (e.g., LDAP), or upgrading software (e.g., package repositories), for example. In cases such as in cloud computing or grid platforms, each machine is itself a constituent of a larger distributed system. Although in this work we have focused on securing a single machine, we believe that the same principles can be applied to a distributed setting by propagating trust across components using secure channels and remote attestation mechanisms. However, we have not yet explored these extensions.

Handling corruption of persistent state. One might argue that the loss of control through the management interface enforced by BROKULOS could hinder the ability for the administrator to recover the system or the user data if bytes are corrupted on disk. In such cases, the procedure to recover a BROKULOS box is analogous to what is typically done to recover a Linux box: If the user data has been tampered with, the administrator can restore it from backups. If instead the software has been corrupted, the administrator can reinstall a clean slate image of the system (if necessary on a different machine).

Improving data availability. In our current design, the administrator has full control over system resources. However, a great deal of security issues can arise due to the accidental deletion of data. To prevent permanent data loss, brokers could, e.g., include delays to enable the administrator to revert the actions that were performed accidentally or even require multiple administrators to authorize more critical tasks. This guard could target all brokers that can cause data to be permanently removed.

6.7 Summary

We introduced the broker security model, a general security model aimed at protecting the confidentiality and integrity of user data from system administration errors. By only trusting administrators for resource availability and not for information security, this model improves data protection with little impact on system manageability. It achieves this property by relying on a layer of *brokers*—trusted programs that mediate access to system objects. We showed that this model is practical for OSes by implementing and evaluating BROKULOS, our proof-of-concept broker-compliant OS. The broker model lays out important principles in the design of untrusted-admin systems. We envision applying it to other software systems (e.g., databases and web applications) and improving the mechanisms necessary to enforce this model (e.g., by reducing the TCB size). By making OSes more resilient to mismanagement threats, the broker security model and BROKULOS have the potential to strengthen users' trust in enterprise platforms. Next, we turn our attention to improving trust in mobile platforms.

Part III

Improving Trust in Mobile Platforms

7 Motivation and Related Work

So far we have focused on trust issues related to cloud and enterprise platforms. We now shift gears to the mobile landscape. A trend of new mobile apps is emerging with strict security requirements. Examples include e-wallet and e-health apps, which require private access to their execution state. However, today's platforms can hardly offer such guarantees. In fact, the complexity of mobile platforms like Android, iOS, or Windows 8 is such that it is almost impossible for these systems to be bug free.

To suit the needs of security-sensitive apps, we propose a design where the OS runs side-by-side with a small sized *trusted runtime system*, both of which run in isolated security domains enforced by hardware. The trusted runtime system provides a reduced TCB environment for hosting security-sensitive components of mobile apps. The specific challenge of building a small trusted runtime system when compared to related systems [MPP⁺08a, MLQ⁺10] comes from the fact that mobile apps do not run native code, but run an intermediate code, which depends on complex runtime engines like the Java Virtual Machine and .Net CLR.

In this part of the thesis, we present the Trusted Language Runtime (TLR). The TLR reduces the size of the TCB, by providing only a minimal number of application runtime services. By this, it is possible to strip off a typical runtime engine from all the components except those that the basic services depend upon. We built a TLR prototype targeting .Net mobile apps. In its implementation, the TLR uses ARM TrustZone technology to protect its state from the OS. In addition, to provide the basic application runtime services, we used the .Net Microframework (NetMF), a lightweight and customizable implementation of .Net for embedded devices.

Before presenting the TLR in Chapter 8, we dedicate the rest of this chapter to frame the problem and to introduce our approach to address it. Firstly, we discuss the limitations of today's mobile platforms in addressing the security needs of emerging apps in the mobile landscape. We then present our idea for enhancing the security of mobile platforms to satisfy those needs, and lay out our plan to realize our idea, namely set our goals, state our assumptions, and characterize the threat model. Lastly, we provide some necessary background and discuss the related work.

7.1 Security Needs of Emerging Mobile Apps

The need for trusted applications on smartphones is greater than ever. As smartphones become the *de facto* personal computing device, people are storing more and more sensitive and personal information on their phones. Unfortunately, the value of this information is starting to make smartphones an attractive target for attacks, including third-party applications with questionable practices [EGC⁺10] as well as outright

malware [Hyp06]. Even more alarming, researchers have demonstrated that today’s smartphones can be subjected to rootkits, which can compromise the OS [BOB⁺10].

At the same time, it is difficult to get a grasp on the security properties provided by existing mobile platforms like iOS, Android, or Windows 8. The fundamental reason for this is their complexity. These systems include a large number of APIs, rely on runtime engines for executing apps, control multiple devices and sensors, and run full blown commodity OSes. Consequently, applications depend on bloated trusted computing bases (TCBs), comprising millions of lines of code (a number comparable in size to that of desktop and server platforms). In such large codebases, the likelihood of outstanding code vulnerabilities is considerable, making it difficult to ascertain the correctness of their implementation.

As a new trend of security-sensitive applications emerges, the need for strong security guarantees becomes critical. For example, e-wallet apps aim to replace physical payment media like credit cards, tickets, or coins involved in transactions. E-health apps focus on carrying personal health records of the device owner and providing for proper protection and access control by health providers. Despite the benefits of these apps for users, service providers such as banks and health authorities could feel reluctant to develop this kind of mobile apps unless mobile devices provide adequate security environments for hosting them. It is thus time to rethink the design of mobile platforms so as to satisfy those needs.

7.2 Hosting Mobile Apps in Trusted Execution Environments

As we discussed in Section 5.4.2, improving the security of desktop and server platforms featuring large TCBs could be achieved by providing *trusted execution environments*. Trusted execution environments have the property of keeping the execution state of security-sensitive applications out of the OS’s reach, giving users the guarantee that the application execution state has not leaked or been corrupted in a surreptitious manner in the event of an OS security exploit. Trusted execution environments are enforced by a *trusted runtime system* running in isolation from the OS in a security domain properly set up by the hardware. What makes this approach reliable, is that trusted runtime systems like TrustVisor [MLQ⁺10] are several orders of magnitude smaller than that of a commodity OS, dramatically reducing the TCB size. Therefore, we propose a similar approach for shrinking the TCB size of mobile platforms.

Figure 7.1 sketches a mobile platform design that enables the execution of security-sensitive mobile apps inside trusted execution environments. This design requires two components: *hardware mechanisms* for isolation enforcement, and a *small trusted runtime system* for managing the execution state of the apps.

To enforce isolation, we propose to use TrustZone technology [ARM09]. Currently available in modern ARM-based devices, this technology enables the processor to run in two protection domains named *normal* and *secure* worlds, where the OS and the trusted runtime system could be hosted, respectively. TrustZone provides separate address spaces between worlds and secure mechanisms for cross-world communication. Be-

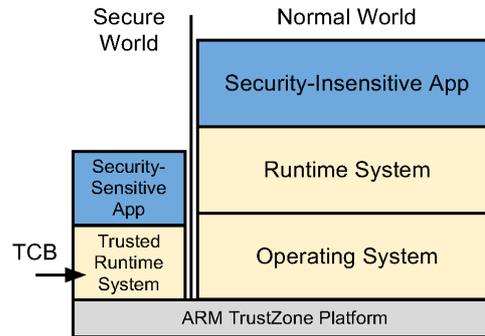


Figure 7.1: Architecture of a mobile platform featuring trusted execution environments.

cause these mechanisms are implemented in hardware, this technique leads to increased efficiency and smaller TCB size compared to using virtualization.

Designing a small trusted runtime system, however, is particularly challenging. Mobile apps are built using high-level languages, which produce a binary in an intermediate language like Java bytecodes or .Net managed code. As a result, they depend on a runtime engine like the Java (or Dalvik) virtual machine or .Net CLR to execute. The trouble is that the runtime engine is typically a large piece of software and, in addition, it depends on the services provided by an underlying OS. The .Net CLR, for example, comprises millions of lines of code. The open question is then whether it is possible to build a small sized trusted runtime system that can execute bytecode / managed code binaries. Note that the related work [MPP⁺08a, MLQ⁺10] did not face this problem, because their focus was primarily on providing execution support for native-code applications.

7.3 Goals, Assumptions, and Threat Model

In part III of this thesis we aim to build a trusted runtime system for mobile platforms—the Trusted Language Runtime (TLR). Using the TLR, application developers can execute security-sensitive applications in trusted execution environments. These environments must provide guarantees of confidentiality and integrity protection of applications’ execution state. Without loss of generality, the TLR provides support for development of .Net mobile applications.

In designing the TLR, we have three high-level subgoals:

1. **Small TCB size:** The trusted computing base (TCB) of the TLR should exclude the operating system and most application code running on the smartphone. None of this untrusted code should be able to interfere with or even inspect trusted code running inside the TLR. The application components hosted within the trusted execution environment should be guaranteed the protection of confidentiality of their execution state.

2. **Ease of programming:** The effort required to build trusted applications with the TLR should be low. Programming in TLR should be as simple as programming any of today's managed code environments such as Java or .Net. We seek to provide intuitive and simple abstractions for application developers to interact with the TLR.
3. **Compatible with legacy software environments:** Running the TLR should not require a radical redesign of today's legacy operating systems or other legacy software running on the smartphone.

We design the TLR under the following assumptions. First, we assume that the hardware platform features the ARM TrustZone technology, which is present in modern ARM processors. Second, we assume that the hardware is correctly implemented, namely the ARM TrustZone technology and the CPU and memory subsystems. Third, we assume the existence of external trusted parties such as certification authorities or online services (e.g., banking web sites) with which the security-sensitive apps could communicate over a secure channel while deployed on the TLR. Lastly, we assume the correctness of cryptographic primitives and algorithms. Note that we make no assumptions whatsoever about the correctness of the OS of the mobile platform.

The TLR is designed to protect the execution state of security-sensitive application components against an attacker with a profile very similar to that of the attacker we consider in Part I and Part II (see Sections 2.3 and 5.3). He can take over the OS and have access to the TLR interface. The interface is provided through specific TrustZone mechanisms. The attacker can reboot the mobile platform and gain access to data residing on persistent storage. He can eavesdrop the network and interfere with the communication between the TLR and third party trusted components located outside the device. In designing the TLR, however, we do not consider side-channel attacks and do not contemplate physical attacks that fall outside the defense capabilities of TrustZone technology, namely attacks that involve disassembling the chip packages of application processors and memory modules.

7.4 Brief Primer on TrustZone and NetMF Technologies

This section provides some background on two technologies we use to implement the TLR: ARM TrustZone, and the .Net Microframework (NetMF).

7.4.1 ARM TrustZone Technology

TrustZone [ARM09] is the name of a hardware technology introduced by ARM in 2008. It provides security extensions that affect the processor-memory subsystem and the System-On-Chip (SoC) layout of ARM architectures. Figure 7.2 illustrates how these extensions affect the software and hardware architecture of a computing platform.

The key feature of TrustZone is the ability to execute code without interference from the OS. There exist two security domains that the processor implements natively, referred

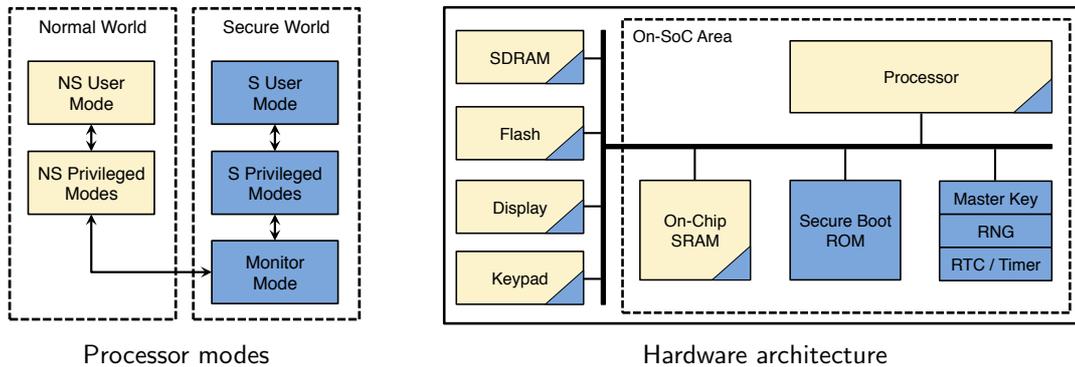


Figure 7.2: Processor modes and hardware architecture of ARM-based device with TrustZone extensions.

to as the *normal world*, where the OS executes, and the *secure world*, where secure services and secure applications run. The security domains have independent memory address spaces. The code running in one of the secure world’s privileged modes (including the monitor mode) has access to the normal world address space. The code executing in the normal world has no access to the secure world address space. To prevent privilege escalation, the processor boots in secure world, and the software must decide whether to remain in secure world or switch to normal world. In a typical bootstrap sequence, the bootstrap code yields to the OS after setting up the secure world state. This is done by exiting secure world and proceeding with the standard OS boot sequence in normal world.

In order for the software of different worlds to communicate, the processor can switch worlds via a narrow interface. Once the system has booted, the OS can invoke a service in secure world by issuing a special software interrupt: the Secure Monitor Call (*smc*) instruction. The processor enters secure world through the *monitor* mode. The code executed in monitor mode is responsible for (i) saving and restoring the execution context of the processor, as it switches worlds, and (ii) sharing data by copying data across worlds or mapping normal world pages into the secure world address space.

TrustZone also allows for the implementation of trusted I/O paths linking secure world software with peripherals. A special processor bit, the *NS* bit, defines the world in which the processor executes, and signals an extra address line that is wired to memory modules and devices. This design enables a software designer to restrict I/O addresses of selected devices (or some of their ports) to the secure world software only. The software designer is also allowed to route interrupts to a world of his choice by programming the interrupt controller and setting interrupt masks accordingly. In this sense, TrustZone makes a step forward over related hardware technologies like Intel TXT [Int], which only provide memory protection capabilities.

All these mechanisms can be used in different ways by system designers. ARM, in fact, does not prescribe a specific software architecture, but suggests multiple system

designs, ranging from setups where two OSEs run side-by-side in independent worlds, to a design where the secure world runs a simple library [ARM09]. Nevertheless, it is clear that, with TrustZone technology, system designers have the means to implement all trusted computing abstractions. Memory curtaining and trusted I/O paths are possible using the memory and peripheral protection mechanisms just described. Trusted boot could be implemented by deploying a secure boot ROM into the SoC; so long as the ROM and the secure world code are trustworthy, the ROM initiates the chain of hash measurements featuring a typical trusted bootstrap sequence and preserves these measurements in the secure-world address space. Remote attestation could be implemented by burning a cryptographic key—a *master key*—into the SoC. This key remains private to secure world software, which could then use it to produce signatures of the software bootstrap measurements. Lastly, sealed storage could be implemented by secure world software using the software measurements and the master key. To strengthen the security of cryptographic operations, the SoC manufacturer could include a random number generator and real time clocks into the SoC (see Figure 7.2).

It is also worth remarking that TrustZone remains largely unused in today’s mobile device landscape. Despite the fact that TrustZone compatible processors (e.g., Cortex A8 and A9) have been available for a while in popular ARM-based mobile devices, device manufacturers presently set up the firmware to disable TrustZone technology, preventing application developers from executing code in secure world. The reasons why this happens are not entirely clear. One possible explanation is that device manufacturers want to monetize this technology by deploying closed-source secure services and getting paid for them. Another explanation involves security concerns. Since the secure world grants access to the whole system, exploits of vulnerabilities in secure world code could compromise the entire OS. We envision that the TLR could contribute to changing the state-of-affairs by providing a secure runtime for hosting trusted applications and open up TrustZone technology to application developers.

7.4.2 Microsoft .Net Microframework

The .Net Micro Framework (NetMF) [net] is an implementation of Microsoft’s .Net Framework optimized for small devices. It enables application programmers to use fully featured development tools like Microsoft Visual Studio and high-level languages like C# to program an embedded system. Examples of such devices include: sensor networks, robotics, GPS navigation devices, wearable devices, medical instrumentation, and industrial automation devices [net10].

The design of NetMF was guided by three main tenets. First, NetMF designers put an emphasis on offering a robust development environment. To this end, the NetMF includes a CLR runtime (the equivalent version of the Java virtual machine in .Net terminology), which provides type system, code execution safety, and garbage collection. The CLR runtime executes applications compiled into *managed code*, an intermediate language akin to Java bytecodes. Application developers can also benefit from high-level language .Net Framework compliance and from a collection of code libraries. Second, NetMF designers tailored NetMF for resource constrained devices. For improved effi-

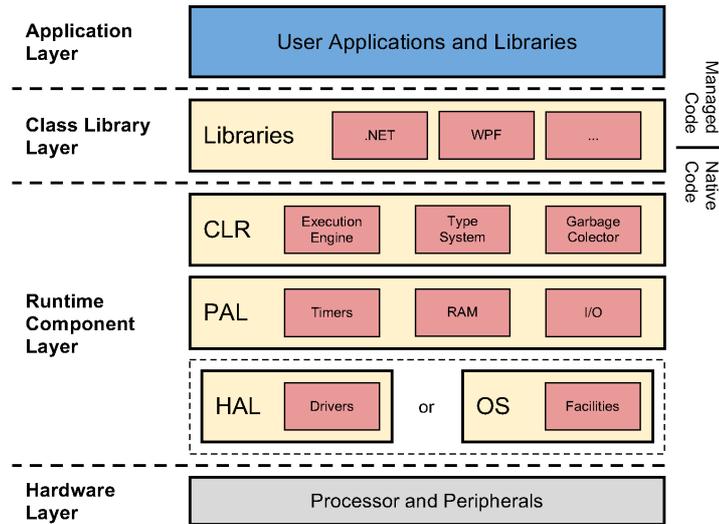


Figure 7.3: Architecture of the .Net Microframework (NetMF).

ciency, NetMF precludes an underlying OS and runs directly on metal. One could say that NetMF consists of a “bootable .Net” that offers the benefits of managed code without requiring a dedicated OS. Internally, NetMF owns all execution, and it includes only the bare system functionality for managing memory, CPU, and peripherals. In its basic setup, NetMF has a small codebase and makes use of only a 250KB memory footprint. Third, NetMF designers focused on customizability for a wide variety of devices. This requirement influenced the internal design of NetMF by having the hardware abstracted. To port the NetMF, the embedded system designer only needs to implement a set of low-level functions that are hardware-specific (e.g., CPU dependent).

Figure 7.3 shows the NetMF hardware and software architecture in more detail:

- The hardware layer consists of the processor and peripherals. The NetMF software can be customized for a number of hardware platforms.
- The runtime component layer includes three components: the Common Language Runtime (CLR), the Hardware Abstraction Layer (HAL), and the Platform Abstraction Layer (PAL). The CLR is the heart of the NetMF. It contains several modules for managed code execution, thread scheduling, memory management, and other system services. It includes a subset of .Net features, such as types, threads, synchronization, timers, reflection, serialization, garbage collection, and exception handling. There are, however, some .Net features not currently supported, the most relevant ones being the lack of multidimensional arrays and templates. The HAL and PAL manage the underlying hardware components. The PAL provides abstractions of the low-level hardware services implemented in the HAL; only the HAL functions need to be ported when customizing the NetMF for

7 Motivation and Related Work

a given hardware architecture. The HAL also includes a piece of bootstrap code responsible for initializing the devices and starting the CLR.

- The class library layer is an object-oriented collection of types that application developers can use to program embedded applications. Classes are implemented in C# and offer multiple services, including cryptographic primitives, graphics, debugging, standard .Net API types, and extensions to specific chipsets.
- The application layer contains the managed-code applications developed by the application programmers for embedded devices.

We use the NetMF codebase as the baseline for implementing a prototype of the TLR.

7.5 Related Work on Improving Trust on Mobile Platforms

There is a large body of work that indirectly helps to reinforce users' trust in mobile platforms by improving the security of OSes and mobile apps. In particular, researchers have paid considerable attention to protecting personal user data (e.g., address book, user photos, password information, GPS location) and preventing its unauthorized access and leakage by proposing novel techniques, such as new access control mechanisms [RKM⁺12] and information flow analysis [EGC⁺10]. The focus of all this work, however, is complementary to ours: while in existing work the OS is trusted, in our work the OS can be compromised and the TLR must provide applications with adequate security protections.

Beyond the scope of mobile devices, previous work has focused on using trusted computing hardware for building systems that provide code and data protection from the underlying OS [GPC⁺03, MPP⁺08a, MLQ⁺10]. Such systems face a tradeoff between security and usability. While some systems depend on a large trusted computing base (TCB) to offer high-level functionality [GPC⁺03], others have small TCBs but offer programming abstractions that are low-level [MPP⁺08a, MLQ⁺10]. The TLR bridges these two extremes by offering a high-level programming abstraction while keeping the TCB small.

Another area of research uses privilege separation for partitioning an application into security-sensitive and security-insensitive components. Typically, these systems expose a partitioning interface at the level of the programming language, and enforce this separation by using a runtime engine [Mye99] or the OS itself [BS04]. In general, however, they still depend on a large TCB, which includes the OS and the runtime. Our work offers a coarser-grained privilege separation at the language level by compartmentalizing an application while significantly reducing the TCB size.

Finally, there is little published work on building systems that use the ARM TrustZone technology for their trustworthy computing needs. One relevant piece of related work proposes to merge the TPM-based primitives found on x86 machines with those found on ARM in order to build a Linux-based embedded trusted computing platform [Win08]. That paper uses a VM-based design and offers a special "TrustZone VM" to run trusted code. In contrast, the TLR avoids the energy and performance overheads that come

with hypervisor-based virtualization systems. Another relevant piece of related work uses TrustZone technology to implement trusted sensors [LSWR12]. Trusted sensors enable mobile apps to obtain guarantees of authenticity and integrity of sensor readings and, therefore, were developed for a purpose different than the TLR's.

7.6 Summary

This chapter focused on the trust issues affecting the mobile device landscape. We have seen that the security guarantees provided by existing mobile platforms do not suit the needs of emerging mobile apps. This lack of guarantees stems mostly from the fact that applications depend on large trusted computing bases (TCB), whose correctness properties are difficult to reason about. To meet the security requirements of mobile apps, we propose a mobile platform redesign offering *trusted environments*, where the runtime state of applications is kept safe from a compromised OS. Trusted environments are provided by a *trusted runtime system* running alongside the OS. ARM TrustZone technology separates the trusted runtime system from the OS. The key challenge, then, is how to design the trusted runtime system with a TCB significantly smaller than that of a typical OS. In the next chapter, we present the Trusted Language Runtime (TLR), our proposal for such a system.

8 Trusted Language Runtime: Enabling Trusted Applications on Smartphones

This chapter presents the Trusted Language Runtime (TLR), a trusted runtime system for mobile platforms. The TLR enables the development and execution of trusted mobile applications while depending on a small TCB.

The rest of the chapter unfolds as follows. First, we present a high-level architecture of the TLR. Then, we dive into the details of the system design, describe the implementation of a TLR prototype, and show the applicability of the TLR in multiple use cases. Finally, we discuss the results of our TLR evaluation and summarize our findings.

8.1 Overview of Trusted Language Runtime

To provide an overview of the TLR, we first state the principles that guided the design of the system. Then, we provide a high-level description of the TLR architecture. Lastly, we describe the development process of mobile applications for the TLR.

8.1.1 Design Principles

In devising the TLR, we followed two key principles: *privilege separation* and *subordinate resource allocation*. The former is widely known; the latter is proposed in this work.

In general, *privilege separation* aims to mitigate the potential damage of a security attack by dividing a program into parts and restricting the privileges of each part to those strictly necessary for performing a particular task. To improve the security of mobile applications, we apply this principle in the TLR design. Specifically, instead of requiring an entire application to be hosted in the secure world (see Section 7.2), the TLR enables hosting different partitions of the application logic in different worlds, while still enabling them to communicate. Thus, an application developer should be able to reduce the amount of code that needs to be trusted for processing security-sensitive data by factorizing the application into security-sensitive and security-insensitive components and hosting them in separate worlds.

The second principle, which we call *subordinate resource allocation*, aims to arbitrate the allocation of memory and CPU resources between the TLR and the OS. It states that the control over the CPU cycles and memory resources consumed by the TLR must be controlled by the OS: the OS must explicitly authorize every resource allocation request issued by the TLR, and is free to revoke the allocated resources at any given time. The OS, however, must have no access privileges over the data associated to the resources granted to the TLR (e.g., read the memory or the CPU registers). Since the

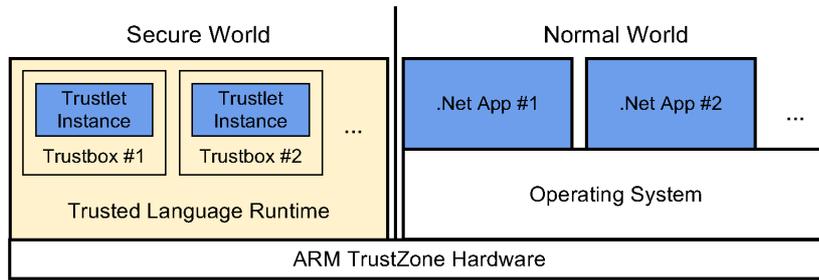


Figure 8.1: High-level architecture of TLR.

OSes are currently designed under the assumption of complete control over the hardware resources of the system, this principle avoids radical changes to OSes and therefore lowers the barriers to deployment of the TLR by the device manufacturers. As a side effect, a compromised OS could deny the TLR access to system resources and by this launch DoS attacks. Although these attacks could be possible, the confidentiality and integrity of applications' state would still be assured, which is our priority.

8.1.2 High-level Design

Figure 8.1 illustrates the TLR's high-level design. The TLR provides two execution environments: an untrusted one where the smartphone's OS and most application software runs, and a trusted one. The code running in the trusted environment is isolated from any code running in the untrusted one. The TLR ensures both integrity and confidentiality for code and data inside the trusted environment. The OS controls the allocation of CPU and memory resources in the trusted world. To enable interaction, the TLR provides a secure communication channel between the two environments.

In the trusted world, the TLR provides a language runtime with minimal library support: in our implementation we use the .Net Micro Framework [net]. We find that a resource-constrained runtime environment offers enough flexibility to accommodate the trusted computing needs of mobile applications while keeping the TCB of the TLR small.

With the TLR, a developer needs to partition a mobile application into two components: a small-sized trusted component that can run on the resource-constrained runtime of the trusted world, and a large-sized untrusted component that implements most of the application's functionality. This partitioning process is similar in spirit to previous work on privilege separation [BS04] and partitioning of applications for improved security in distributed systems [CLM⁺07]. To allow for this partitioning, the TLR's programming model provides four primitives:

- 1. Trustbox.** A trustbox is an isolation environment that protects the integrity and confidentiality of any code and state hosted inside it. This means that the smartphone's

OS (or any untrusted application code) cannot tamper with the code running in a trust-box nor inspect its state.

2. Trustlet. A trustlet is a class within an application that runs inside a trustbox. The trustlet specifies an interface that defines what data can cross the boundary between the trustbox and the untrusted world. The .Net runtime's use of strong types ensures that the data crossing this boundary is clearly defined.

3. Platform identity. Each device that supports the TLR must provide a unique cryptographic platform identity. This identity is used to authenticate the platform and to protect (using encryption) any trusted application and data deployed on the platform. Our implementation uses a public/private key pair. Access to the private key is provided solely to the TLR, which never reveals this key to anyone.

4. Seal/Unseal data. These abstractions serve two roles: (i) allow a trustlet to maintain state across reboots, and (ii) enable a remote trusted party (i.e., a trusted server) to communicate with a trustlet securely. Sealing data means that data is encrypted and bound to a particular trustlet and platform before it is released to the untrusted world. The TLR unseals data only to the trustlet identity and target platform that were specified upon sealing. The trustlet's identity is based on a secure hash of its code (e.g., SHA-1). Both the platform and trustlet identities are specified at seal time. To recover sealed data, the TLR decrypts it using the platform key, and checks that the hash of the calling trustlet matches the hash of the trustlet that originally sealed the data.

8.1.3 Development Scenario

To build a trusted mobile application with the TLR, a developer would typically perform the following three steps:

1. Determine which part of an application handles sensitive data. To define a trustlet, the developer identifies the application's sensitive data, and separates the program logic that needs to operate on this data into the trustlet. The developer carefully defines the public interface to the trustlet's main class, as this interface controls what data crosses the boundary between the trusted and untrusted worlds. A trustlet may use many helper classes, and in fact may even consist of multiple assemblies, yet there is only one class that defines the trustlet's boundary. Once all necessary classes are compiled into assemblies, the developer runs a TLR post-compilation tool for creating a package that contains the closure of the assemblies, and a manifest that contains meta-data information.

2. Seal the sensitive data by binding it to the trustlet. To make sure that a destination mobile platform is configured with the TLR and running the trustlet, before sending out sensitive data, a developer can *seal* the data to that specific mobile platform

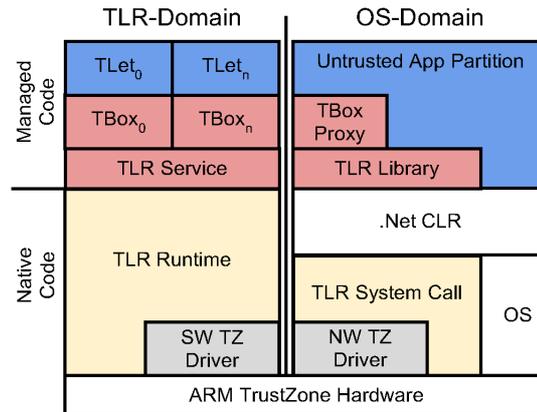


Figure 8.2: Component diagram of the entire system with the TLR. The components of the same layer are colored with the same color.

and trustlet identity. The data can only be *unsealed* if both conditions hold. The sealing mechanism allows the application to store trustlet data across multiple sessions in persistent storage, and it allows external parties (e.g. a trusted service) to ensure that sealed data can only be accessed on platforms it trusts.

3. Deploy trustlet and sealed data to the smartphone and run them inside of a trustbox. To ensure that the trustlet state is protected at runtime, the developer instantiates a trustbox by providing the trustlet’s manifest. At this point, the TLR loads the trustlet’s assemblies and creates an instance of the trustlet main class. The resulting object constitutes the runtime state of the trustlet until the application destroys the trustbox. To allow the application to interact with the trustlet, the application obtains from the TLR a special *entrypoint* object, which is a transparent proxy to the trustlet interface. Whenever the application invokes methods on the entrypoint, the TLR transparently forwards these calls to the trustlet main object.

8.2 Design of Trusted Language Runtime

In this section, we describe in detail the internals of the TLR and discuss its key design decisions. We start with a holistic view of the TLR design, and then focus on its most relevant components.

8.2.1 Internals of the TLR

Figure 8.2 shows a detailed view of the TLR architecture. It is structured as a software stack and spans two security domains that are enforced by the ARM TrustZone technology: the *TLR-domain* and the *OS-domain*. The TLR-domain is mapped to the secure world and hosts the TCB of the system: the *TLR core* components and the trusted

application partitions. The OS-domain is mapped to the normal world and hosts the untrusted system components: the OS, the *TLR stubs*, and the untrusted application partitions. Internally to the TLR, the stack consists of multiple layers, each of them containing a subset of TLR core components (in the TLR-domain) and TLR stubs (in the OS-domain). As shown in Figure 8.3, the stubs and core components of each layer communicate using specific message formats. In this communication, stubs and core play the role of client and server, respectively. To exchange messages, the components of a layer rely on a message passing service provided by the contiguous bottom layer. Because TrustZone provide hardware support for memory isolation, the TLR runtime lives in the address space of the trusted world and cannot be accessed from the untrusted world.

The TLR software stack comprises four layers: *application layer*, *trustbox layer*, *runtime layer*, and *trustzone layer*:

- The *application layer* corresponds to the mobile application, which is split into a trusted and an untrusted partition. The trusted partition is modeled as a set of trusted classes (*trustlets*) instantiated in special sandboxed objects (*trustboxes*). The untrusted partition interacts with the trusted partition using a method call abstraction implemented by the trustbox layer.
- The *trustbox layer* manages the state of trustboxes in the TLR-domain and handles the communication between the untrusted application partition and the trustlet instances living in trustboxes. In the TLR-domain, the state of trustboxes is held in special containers, which are managed by a dedicated service. The OS-domain communicates with the service using trustbox proxies and the TLR library, both playing the role of stubs. A new service is spawned whenever an application is executed. The newly launched service is bound to the application process and is responsible for managing the trustboxes instantiated by the application process. A service is destroyed as soon as its respective application process terminates.
- The *runtime layer* manages the lifecycle of services. In the TLR-domain, the TLR runtime manages the thread state of existing services, executes the managed code of services and trustlet code, and serves their memory allocation needs. The TLR runtime is coordinated by a system call that is included in the OS. This OS extension binds each service located in the TLR to a local application process. The system call receives messages from the trustbox layer and forwards them to the TLR runtime through the trustzone layer.
- The *trustzone layer* masks the low-level TrustZone mechanisms under a simple message passing abstraction. In particular, it handles issues related to world switching and interrupt handling. To handle these issues, each domain implements specific trustzone drivers.

In the following sections, we explain in detail how the TLR works internally. We start by clarifying the programming model offered by the TLR. Then, we focus on

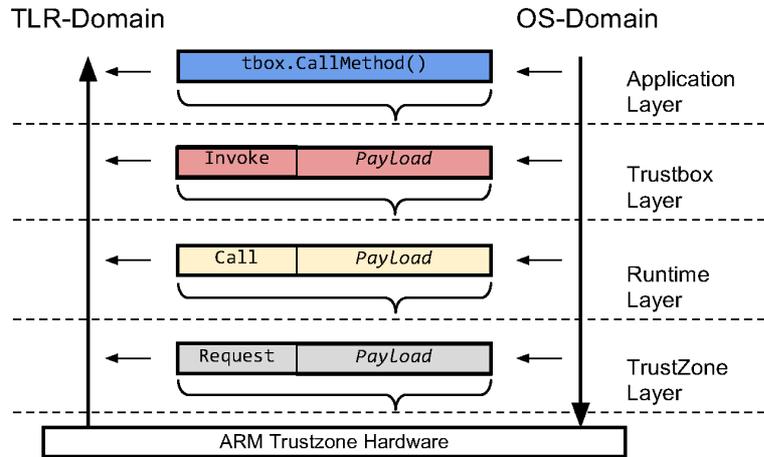


Figure 8.3: Messages exchanged within and across the layers of the TLR software stack.

the mechanisms that implement this programming model: application runtime support mechanisms, memory management issues, and seal and unseal primitives. Lastly, we provide an overview of the TLR bootstrap sequence.

8.2.2 Programming Model

The programming model of the TLR defines programming abstractions and tools that developers must use for building their applications, and therefore it concerns the application layer of the TLR stack (see Figure 8.2). To facilitate the development workflow, we aimed to provide familiar abstractions and tools that can easily integrate into developers' rich programming environments.

To present the programming model, we start by firstly focusing on the implementation and then on the compilation and linkage phases of the development workflow.

Implementing an Application

To implement a TLR application, a developer performs four steps: (i) specify the security-sensitive logic of the app in a *trustlet* class, (ii) instantiate the trustlet class in a *trustbox* container, (iii) interact with the trustlet instance through *transparent proxies*, and (iv) validate the identity and integrity of trustlet instance before giving it access to sensitive data by using *seal* and *unseal* primitives. We cover each of these steps in turn. To help the explanation, we refer to Figure 8.4, which contains code snippets of an example e-banking application. (See Section 8.4.1 for more details on this code.)

1. Specify the security-sensitive logic: The security-sensitive logic of the app must be enclosed in a trustlet class. A trustlet class consists of a unit of data (fields) and code (methods) that can be instantiated in a trustbox container. When instantiated, the data

Trustlet Interface	Trustlet Manifest
<pre>public interface ITanWallet : IEntrypoint { public void Load(Envelope tanLst); public Tan GetTan(long id); }</pre>	<pre><trustlet name="TanWallet"> <interface name="ITanWallet" /> <implementation name="TanWallet" /> </trustlet></pre>
Trustlet Implementation	Snippet of Main Class
<pre>public class TanWallet: ITanWallet, Trustlet { private TanList _tanLst = null; public void Load(Envelope tans) { try { _tanLst = (TanList) this.Unseal(tans); } catch(Exception e) { throw new Exception("Cannot " "unseal TAN list."); } } public Tan GetTan(long id) { Tan tan = _tanLst.Search(id); if (tan == null) { throw new Exception("ID invalid."); } else { return tan; } } }</pre>	<pre>// setup the TAN wallet trustlet in a trustbox Trustbox tbox = Trustbox.Create("TanWallet.pkg"); // obtain a reference to trustbox entrypoint ITanWallet twallet = (ITanWallet) tbox.Entrypoint(); // load the TAN list issued and sealed by bank twallet.Load(myTanLst); // obtain a TAN with id requested by bank Tan tan = twallet.GetTan(id);</pre>
	Snippet of Third Party Service
	<pre>// the bank generates TAN list for customer TanList newList = customer.GenTanLst(); // seal the list Envelope sealedLst = Trustlet.Seal(customer.PlatformID(), Trustlet.Hash("TanWallet.pkg"), newList);</pre>

Figure 8.4: Code sample of a TLR application (written in C#).

fields and the method code of the trustlet instance are inaccessible to the OS-domain; only a method interface is exposed. The developer creates a trustlet by defining its interface and implementing the class. The interface must inherit from the `IEntrypoint` interface. The trustlet main class must inherit from the `Trustlet` class and implement the newly defined trustlet interface. The public methods that implement the interface enable data to cross the barrier between the trusted and untrusted worlds; the strongly-typed nature of the TLR runtime makes it simple to reason about what kind of data is crossing the barrier of the trustlet interface. This is important because the programmer must be careful not to let any sensitive data protected by the trustbox leak out into the untrusted world. To indicate the class and interface of a trustlet, the developer also creates a manifest. Figure 8.4 provides sample code of the interface, class, and manifest of a trustlet implementation.

2. Instantiate a trustlet inside a trustbox container: After implementing the trustlet, the developer must instantiate the trustlet class in a secure container maintained by the TLR—a *trustbox*. The TLR library provides the `Trustbox` class, which allows for controlling the lifetime of a trustbox and of the object hosted on it, i.e. the trustlet instance. To create a trustbox, an application invokes the `Create` method of this class. This method takes as input the trustlet manifest, and creates a new trustbox holding a new instance of the trustlet class referred to by the manifest. When the trustlet instance is no longer required, the application invokes the `Destroy` method of the trustbox class to clean up the runtime state of the trustlet and release all of its resources.

3. Interact with the trustlet instance: During the lifetime of a trustbox, the untrusted application partition is allowed to interact with the trustlet instance contained in the trustbox by invoking methods. Since the trustlet instance and the untrusted application partition reside in separate domains, the method calls must be routed across domains. To make this process transparent to the developer, the TLR library returns a proxy object with a method interface compatible with the trustlet. To obtain a transparent proxy to the trustlet endpoint, the developer must call the `Entrypoint` method of the trustbox reference returned by the `Create` method of the `Trustbox` class. As we explain below, the proxy code is generated during the compilation phase.

4. Validate trustlet identity and integrity: Since arbitrary trustlet code can be instantiated inside trustboxes, third parties relying on the correctness on the trustlet code must have the ability to validate the identity and integrity of the trustlet instances before uploading security-sensitive data into the trustbox. To allow for this validation, the TLR provides `Seal` and `Unseal` primitives. Sealing is a form of encryption that binds the encrypted data to a specific trustlet running on a specific system. To accomplish this, each unique smartphone has a public/private keypair we call the *platform id*. This platform id is used in combination with the secure hash of the trustlet codebase to identify a particular instance of a trustlet. `Seal` takes three inputs: 1) the object to be sealed, 2) the public key of the target platform id, and 3) a secure hash of the target trustlet. `Seal` returns an envelope, which consists of the serialized object concatenated with the trusted hash value, encrypted using the platform id public key. `Unseal` decrypts the envelope and then returns the original data only if the hash value of the currently running trustlet matches the envelope hash value. The envelope can only be decrypted using the platform id private key. Thus, unseal can validate the identity and integrity of a trustlet.

Compiling and Packaging the Application

Once the app is properly implemented, it must still be compiled and packaged. In addition to the standard tool chain operations, two additional steps are required using tools we developed. First, using a pre-compilation tool, we generate transparent proxies for trustlet instances. Proxies are responsible for marshaling the parameters and return

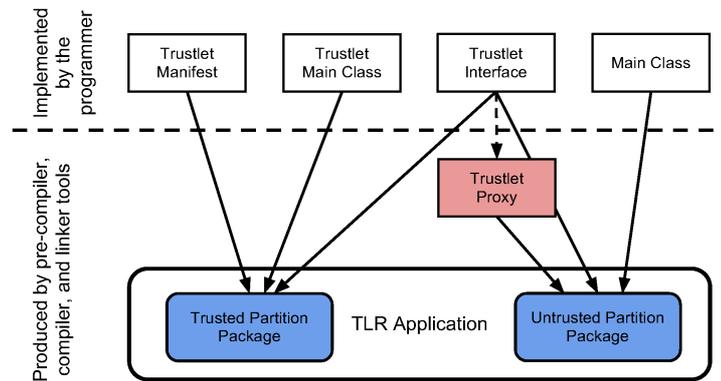


Figure 8.5: Development workflow of a TLR application.

values of the trustlet method call invocation, and for encoding them into trustbox messages that are sent to the TLR’s trustbox service. Second, using a packaging tool, we bundle the code of the trustlet code and the manifest into a single package. This package contains the closure of the trustlet code, i.e., all classes that a trustlet depends upon. This package is also signed in order to allow for the validation of the trustlet’s identity and integrity during unseal operations. Figure 8.5 illustrates this process.

8.2.3 Trustbox Management

After compiling and packaging an app, users can execute it on the smartphone. The TLR automatically manages the trustboxes created by the app, loads and instantiates trustlet code in the trustboxes, and routes method calls across worlds. All these tasks are performed by the components of the trustbox layer, namely the trustbox service in the TLR-side and the trustbox library at the OS-side (see Figure 8.2).

The trustbox layer needs to address three issues. First, trustboxes need an identity so that trustbox requests issued by an application are routed to the intended trustbox. Second, trustboxes must protect their trustlet instances from the surrounding environment and vice-versa: (i) only the code specified in the manifest is allowed to execute in a trustbox, and (ii) the trustlet code execution must be confined to the trustbox domain so as to prevent misbehaved or buggy trustlet code from interfering with other trustboxes and with the trustbox service code. Third, the runtime state of the trustbox (which includes the trustlet instance) must be consistent across invocations of the trustlet methods and subsequent world switches.

To satisfy these requirements, the TLR maintains dedicated *service threads* for keeping track of the trustboxes instantiated by application processes. To host trustboxes, a service thread uses *trustbox holder* data structures (see Figure 8.6). A trustbox holder contains a trustbox ID and points to a sandbox object. The sandbox is a container that maintains the state of a trustlet instance, handles loading of trustlet classes into memory, enforces isolation across trustbox domains, and provides an interface for invoking

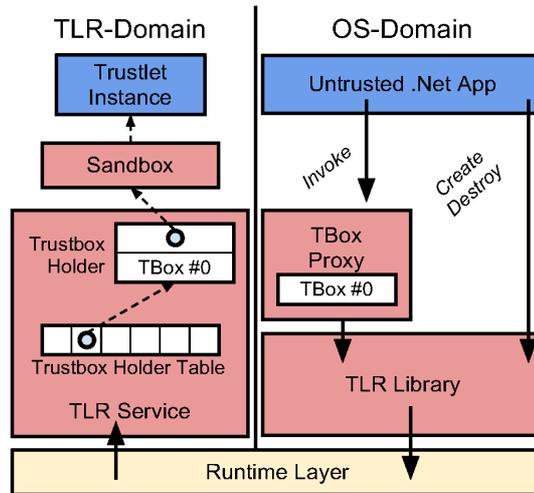


Figure 8.6: Details of the trustbox layer.

methods of the trustlet instance. (In .Net, the sandbox object can be implemented with an `AppDomain` object.) Since TLR preserves the state of a service thread throughout the lifetime of its corresponding application process, the runtime state of trustboxes is consistently maintained between method invocations.

Based on these mechanisms, the trustbox layer handles the three main events of the trustbox lifecycle as follows:

- Trustbox creation:** When the application requests the creation of a trustbox, the TLR library encodes this request in a message and sends it to the respective service thread running in the TLR-domain. The service thread then performs the following steps: 1) creates a new trustbox holder, containing a new ID and a sandboxed container, 2) computes the hash of the trustlet code specified in the manifest, 3) loads the trustlet classes into the sandbox, and 4) creates an instance of the trustlet's main class. Afterwards, the TLR returns a reference to the trustbox (the trustbox ID), which can be used for future interactions with the trustbox.
- Trustbox invocation:** When the application calls the `Entrypoint` method on the trustbox reference, the TLR library creates a transparent proxy and returns it to the untrusted part of the application. Later, whenever the untrusted application invokes a method of the proxy, the proxy forwards a method invocation request to the TLR-side service thread. There, the request is decoded, and the corresponding method is invoked on the trustlet instance living inside the sandbox container of the referred trustbox. The return data produced by the method is forwarded back to the proxy, and returned to the application. This process is entirely hidden from the application.

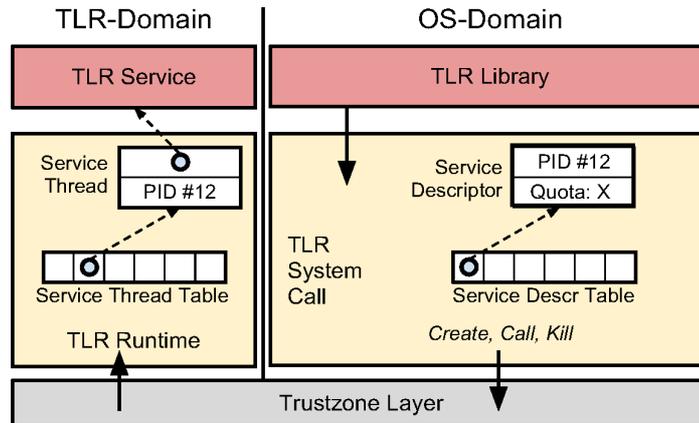


Figure 8.7: Details of the system layer.

- Trustbox destruction:** Finally, destroying a trustbox triggers a request to the service thread for releasing all resources associated with the respective trustbox holder. This will result in discarding the internal state of the trustbox, i.e., the trustlet instance. In order to save any state persistently across instances, the developer can make use of the seal primitive to encrypt the relevant state and have the application store it persistently.

In order for the TLR library code of an application process to communicate with the corresponding service thread, the trustbox layer uses the transportation services provided by the runtime layer, which we describe next.

8.2.4 Runtime Support

In the previous section, we focused on the trustbox layer. We saw how it splits the application state between a server thread, which lives in the TLR and manages the trustbox state, and the application process, which hosts the security-insensitive app state. In this section, we discuss how the runtime layer (see Figure 8.2) provides the underlying runtime support infrastructure that makes this splitting possible.

There are several issues that the runtime layer must handle. First, since trustlets are encoded in managed code, the TLR runtime must interpret and execute managed code (rather than simply running native code). Second, given that multiple applications can execute concurrently in the system, the TLR runtime must be multitasked. This entails that multiple service threads may live simultaneously in the TLR and, naturally, that each service thread must be unambiguously bound to the respective application process running in the OS-domain. Finally, since the runtime layer provides a message delivery service to the trustbox layer, an appropriate interface must be devised for this service, preferably without requiring significant changes to the OS.

To address these issues, the runtime layer implements several mechanisms in both domains (see Figure 8.7). In the TLR-domain, the TLR runtime includes subcomponents

that allow for the execution of managed code: a managed code execution engine, a type system enforcer, and a garbage collector. The TLR runtime then sits on a loop waiting for incoming requests from the OS-domain and executing the managed code required for serving them. Since incoming requests may refer to different application processes, the TLR runtime maintains independent service threads. To enable service threads to execute (managed code) trustlets, service threads are implemented as user-level managed code threads managed by the TLR runtime. To bind a service thread to an application process, the TLR annotates the descriptor of the service thread with the PID of the application process. This annotation enables the TLR runtime to route an incoming request to the service thread based on the PID of the calling application process.

In the OS-domain, the OS (via the TLR system call) coordinates the servicing of trustbox requests by TLR runtime. An application process sends requests to the TLR runtime through a special TLR system call added to the OS. These requests are then forwarded to the service thread bound to the calling application process. Since the TLR runtime is not aware of application processes' lifetimes, the OS must instruct the TLR runtime to create and destroy service threads according to the needs of application processes.

To accommodate the communication needs within the runtime layer, the OS can exchange three messages with the TLR runtime:

- **Create service thread:** The first time a TLR system call is issued by an application process, no service thread for that process exists, and so it must be created. To keep track of which application processes are bound (i.e., are associated with a service thread) the OS maintains a descriptor table. To bind an application process, the OS issues a “create service thread” request to the TLR runtime. The TLR runtime creates a new service thread and annotates it with the caller’s PID contained in the request. Once the request has been served, the TLR returns to the OS, which continues executing the system call. Since the system call is invoked for sending a message, normally a “call service thread” request ensues.
- **Call service thread:** This operation is meant to forward the trustbox message requests received through the system call interface to the server thread of the calling application process. After making sure that the calling process is bound, the OS issues a request to the TLR runtime. The TLR runtime retrieves the trustbox message from the payload of the request, puts the message in a queue, and resumes the execution of the respective service thread. The service thread retrieves the request from the queue and processes it in the trustbox layer. After it finishes serving the request, the service queue executes a special managed code call to signal the TLR runtime of the service completion. The TLR runtime then returns to the OS, which concludes the system call.
- **Kill service thread:** This operation is issued by the OS to terminate a service thread and free its resources. One way to trigger this operation is through the TLR system call. Typically this is done by an application that has destroyed all its trustboxes. The other way is by the OS, which periodically kills service threads

of terminated application processes. To kill a service thread, the OS simply sends a request to the TLR runtime and updates the local OS data structures. The TLR runtime releases all resources and data structures allocated to the service thread.

All messages exchanged within the runtime layer are carried by the trustzone layer, which we cover next.

8.2.5 Cross-world Communication

The communication between the TLR-domain and the OS-domain is handled by the bottommost layer, which is the trustzone layer (see Figure 8.8). This layer handles the low level details of TrustZone technology and provides a simple message passing interface that enables the runtime layer to exchange messages across worlds.

Specifically, the trustzone layer has to deal with two main issues. First, since the processor can only execute in one of the worlds—the normal or the secure world—sending a message across domains requires a world context switch. This operation must be carefully implemented: it must be efficient without creating security breaches by exposing the TLR state to the normal world. Second, it is necessary to handle interrupts that could be triggered while the processor is in secure world. According to the principle of subordinate resource allocation (see Section 8.1.1), the OS must retain control of the system resources, and therefore interrupts must be routed to the OS. Thus, whenever an interrupt is fired in the secure world, the TLR should cause a context switch to normal world, and hand over the control to the interrupt handler of the OS. This policy, however, has two potentially problematic consequences. First, because interrupts can fire in the middle of a TLR runtime call, the OS could resume its execution without the TLR call producing a result. Second, since the OS might decide to schedule a different process for the next time slot, the TLR may need to switch service thread contexts to reflect the process switch that occurred in the OS-domain.

To satisfy these requirements, we adopt the mechanisms depicted in Figure 8.8. For world switching, the SW trustzone driver maintains a data structure with two *world descriptors*: one containing the snapshot of NW registers (i.e., the state of the application process that made the TLR runtime call) and another containing the snapshot of SW registers (i.e., the context of the native-code execution thread of the TLR runtime before leaving SW). Switching worlds, then, can be done efficiently by saving / restoring the processor registers into / from the corresponding world descriptors and toggling the N bit. To guarantee the security of world switching, the world descriptors are kept in memory pages restricted to the SW. In addition, a world switch can be triggered in only two ways: by interrupts being fired in secure world, causing an *asynchronous* world switch, or by a `smc` instruction being issued, causing a *synchronous* world switch. The `smc` instruction can be executed because either (i) the OS makes a TLR runtime call, or (ii) the TLR returns from a TLR runtime call. Such a narrow interface helps reduce the attack surface of the TLR.

With respect to interrupt handling, we need to address the consequences that the disruption of ongoing TLR runtime calls can have on the semantics of TLR exit and

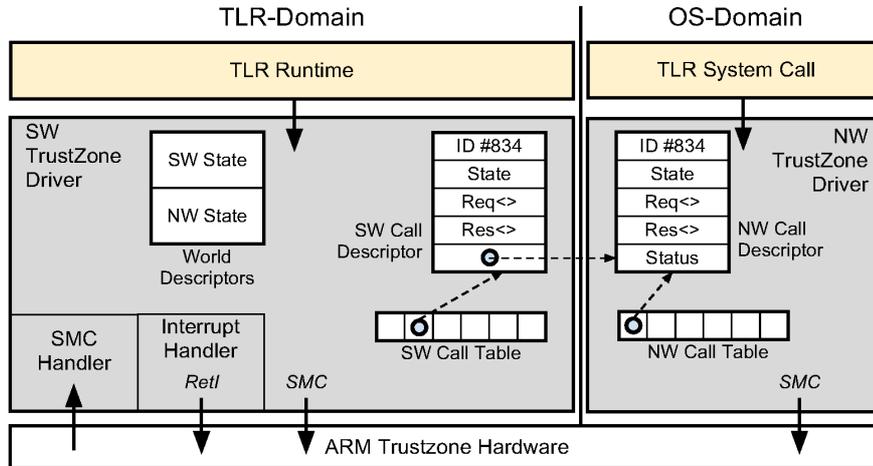


Figure 8.8: Details of the trustzone layer.

enter events. To properly handle a TLR exit event, the OS must be able to detect whether the TLR has exited because the TLR runtime call has terminated or because it has been interrupted. In the first case, the OS can fetch the return value and yield to the application process, otherwise the OS must keep entering the TLR until the TLR runtime call has been entirely served (or a timeout expires). To properly handle a TLR enter event, the TLR must be able to detect whether the current request concerns the service thread that was last executing before the TLR exited or if it concerns a new service thread (because a process switch occurred in the OS-domain). In the first case, the TLR runtime can simply resume the execution of the prior service thread; otherwise, it needs to switch service threads.

To address these issues, we implement a mechanism that enables both worlds to synchronize on the current state of ongoing TLR runtime calls. Essentially, the trustzone drivers of each world maintain a table of descriptors, one for each TLR runtime call. Each descriptor contains a call ID, input parameters, output parameters, and the state of the call. In order to shield the secure world from interference from the normal world, the table of descriptors is replicated in both worlds (see Figure 8.8). However, in certain situations the SW trustzone driver must access the NW descriptor table, namely for reading the input parameters, for writing the output parameters, and for updating the state of an ongoing call. Since the NW trustzone driver has no access to the SW domain, it is the access by the SW trustzone driver to the NW descriptor table that enables them to synchronize on the current evolution of the calls. This synchronization is regulated by the evolution of the state value according to the state machine represented in Figure 8.9 and described next:

- **Init state:** To initiate a new TLR runtime request, the NW trustzone driver creates a new call descriptor, fills out the state field with an `Init` value, and updates the input parameters. It then enters the TLR using the `smc` instruction.

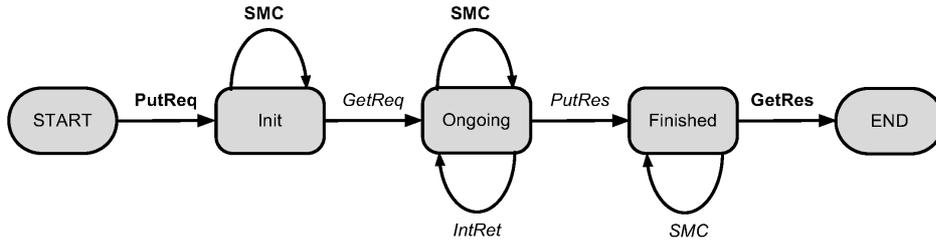


Figure 8.9: State machine of a TLR call as implemented in the trustzone layer. Events in bold take place in the NW, and events in italic in the SW.

Whenever it is in an `Init` state, the TLR creates a new descriptor for the call, copies the input parameters, and changes the state of the call to `Ongoing`.

- **Ongoing state:** In this state, the TLR has started to serve the call. If interrupts occur and control returns to the OS-domain, the NW trustzone driver keep issuing `smc` instructions until the call state turns `Finished`, indicating that the call has been served. This state change occurs after the TLR runtime call has finished and the results have been copied to the NW trustzone descriptors.
- **Finished state:** During this state the SW trustzone driver keeps issuing `smc` until the NW trustzone driver has retrieved the output parameters from the descriptor and terminated the call. At this point, both worlds can release the descriptors allocated to the call.

The trustzone layer hides all these implementation details from the runtime layer under a simple message delivery abstraction (see Figure 8.9). Primitives `PutReq` and `GetRes` provided by the NW trustzone driver allow for sending a request and reading the response. The complementary primitives `GetReq` and `PutRes` are provided by the SW trustzone driver, and allow for reading the request and issuing a response.

8.2.6 Memory Management

To serve the memory needs of the TLR, the system reserves a region of physical memory for the TLR. The TLR uses this memory region for keeping its internal state and the execution state of service threads. To prevent access to this memory by untrusted OS-domain components, the permissions of this region's memory pages are set to SW access only. In our current design, the amount of reserved memory for the TLR is statically defined by a boot parameter (e.g., 10% of the physical memory). However, to reduce memory fragmentation, a dynamic allocation mechanism could be devised in which the OS could grant and withdraw memory pages to and from the TLR according to the current workload demands. Such a memory management policy could be implemented using the virtual memory mechanisms provided by TrustZone technology.

Since the OS is in charge of controlling the system resources, it is possible for the OS to fine-tune the memory allocation policy of the TLR for server threads. Namely, the

OS can specify memory *quotas* for service threads (see Figure 8.6). The TLR runtime enforces a quota limit whenever a server thread allocates memory. If the max quota has been reached, an exception is thrown and forwarded to the application.

8.2.7 System Boot

As described in Section 7.4.1, when an ARM CPU supports the TrustZone feature, the processor boots in secure mode and runs the secure bootloader. Our bootloader is responsible for loading the TLR image into memory and checking its integrity. Next, the secure bootloader hands off to the initialization code within the TLR runtime. After the TLR initialization code finishes, it uses a mode switch instruction to exit secure mode, at which point the untrusted world bootloader is invoked and the standard OS boot sequence is executed.

8.3 Implementation

We implemented a prototype of the TLR for a real TrustZone-compatible hardware testbed and leveraged existing open source software in the TLR implementation.

8.3.1 Hardware Testbed

Finding a TrustZone-compatible hardware testbed was not easy. Although the ARM TrustZone technology is prevalent in modern ARM-based SoCs, in most devices this technology is locked and cannot be used by application developers. Since their manufacturers program the device firmware to force a secure world exit before booting the OS, it is impossible for application developers to execute code in the secure world. This is the case for popular smartphones (e.g., iPhone 4 and Samsung Galaxy III) and widely available developer kits (e.g., Panda [Pan] and Beagle [Bea] boards). Although a few exceptions exist where boards boot the OS in the secure world, manufacturers impose non-disclosure and lock-in restrictions (e.g., Freescale boards [Fre]).

Given these difficulties, we adopted a less than ideal development: the Tegra 250 Dev Kit [Teg] manufactured by NVidia. This board is equipped with dual-core Cortex A9 processors at 1GHz, 1GB of RAM, 512MB of flash memory, and multiple peripherals. Since the processor boots the OS in secure world, this allows us to override the secure world environment. However, this board does not allow to us to flash a unique key in the board's secure ROM. Therefore, we cannot implement a platform ID in hardware. Moreover, the primary boot loader is closed source, preventing us from installing the secure-world setup code early in the first level bootstrap stage.

In our implementation, we address the first limitation by simulating the platform ID credentials in software and configuring them in the secure-world setup stage. To address the second limitation, we had to boot the TLR using a customized second-level bootloader—u-boot [U-b]—resulting in the unnecessary inclusion of the first-level bootloader in the TCB. These shortcomings, however, are not fundamental and could be overcome by adopting an open and fully featured TrustZone board.

8.3.2 Software Implementation

Before implementing the TLR prototype, we had to agree on the programming language and runtime environment on which mobile applications should be built and executed. The TLR targets .Net applications programmed in C# and compiled to .Net managed code. To take advantage of open source software, the runtime environment is based on the Linux kernel 3.5.1 and Mono [Mon] 2.6.7, an open source .Net framework implementation. We then had to implement (i) a few extensions to the runtime environment codebase (the OS-domain), and (ii) the components of the TLR (the TLR-domain).

Regarding the OS-domain, we created a TLR library for Mono and modified the Linux kernel in two ways. First, we extended the kernel with a TLR system call implementation. Second, and less obviously, we had to port the Linux kernel so that it could bootstrap in normal world. On the Tegra 250 Dev Kit, the Linux kernel booted in secure world, but it was not ready to start in normal world: by exiting secure world before jumping to the kernel bootstrap routine, the kernel would eventually execute instructions that are illegal in normal world and hang. After we identified the illegal operations, we fixed this issue and made several changes to the kernel: (i) configured the interrupt masks appropriately, (ii) disabled some cache control registers, and (iii) removed some processor specific initialization code. Some of these operations were included in the TLR setup code.

With respect to the TLR-domain, our implementation covered the TLR and the bootloader. To build the TLR we leveraged the codebase of the .Net MicroFramework (NetMF) v4.1 [net]. As we saw in Section 7.4.2, the NetMF is a much smaller version of the standard .Net Framework, specifically designed for resource constrained devices, and highly customizable. From its codebase, we borrowed the CLR and PAL code, and implemented the remaining components of the TLR stack in the HAL layer and application layers (see Figure 7.3). To customize the NetMF, we used the NetMF porting kit [net07]. As for the bootloader, we customized u-boot to initialize the TLR in secure world and jump to the OS in normal world. To initialize the TLR, u-boot simply loads the TLR binary and jumps to the TLR binary's entrypoint.

8.4 Use Cases

In addition to the TLR prototype, we implemented mobile applications that illustrate how the TLR could improve security in four use cases: one-time password generation, user authentication, secure mobile transactions, and access control to sensitive data. In this section, we present these use cases. For each of them we describe the motivation, state the security goals, and present the security protocols of the application. To describe the cryptographic protocols of the applications, we use the same notation as in the protocols of Excalibur (see Section 4.3). The assumptions and threat model described in Section 7.3 remain valid in this section.

Application Model	Trustlet Interface
	<pre>interface ITanWallet : IEndpoint { void LoadTanLst(Envelope tanlst); Tan GetTan(long index); }</pre>
Protocols	
Setup:	1. Bank→Device: $\text{seal}([\text{TANLST}], Tlet, Dev)$
Query:	1. Svc→Device: $index$ 2. Device→Svc: $\text{TANLST}(index)$

Table 8.1: Use Case 1: Online banking transfers.

8.4.1 One-time Passwords

Our first use case shows how the TLR could protect the generation of one-time passwords (OTP) on smartphones. OTPs are often used to improve security by providing an additional authentication factor, for example in online banking.

To authorize online transfers, banks normally issue lists of OTPs called *Transaction Authentication Numbers* (TANs) [tan] that they send to their customers. Whenever a customer performs an online transfer, the bank specifies an index into the TAN list and asks for the TAN associated with that index. In addition to typing a personal password, a customer must respond with the correct TAN, otherwise the transaction is aborted. To reduce the chance of TAN list compromise (e.g., via browser malware), banks usually write down the TAN list on a plastic card, and send that card to the customer over an out of band channel (e.g., physical mail).

This method, however, incurs an additional burden for customers, who now have to carry along an additional token. Instead, banks could take advantage of the TLR to securely store *digital* TAN lists on customers' smartphones. In other words, the physical tokens containing the TAN list could be replaced by a mobile app—a *TAN list trustlet*—that (i) keeps track of the TAN list on a customer's smartphone, and (ii) provides an interface for querying a TAN based on TAN indexes. The security properties of such an application would be equivalent to the physical token approach.

Figure 8.1 illustrates how such an app could be built. The bank creates the trustlet code (which must be trusted) and seals TAN lists on a per-customer basis so that a TAN list can only be unsealed by the bank's trustlet running on the TAN list owner's phone (protocol Setup). When the online banking service queries the user for a given a TAN, the user feeds the requested TAN index into the TAN list mobile application.

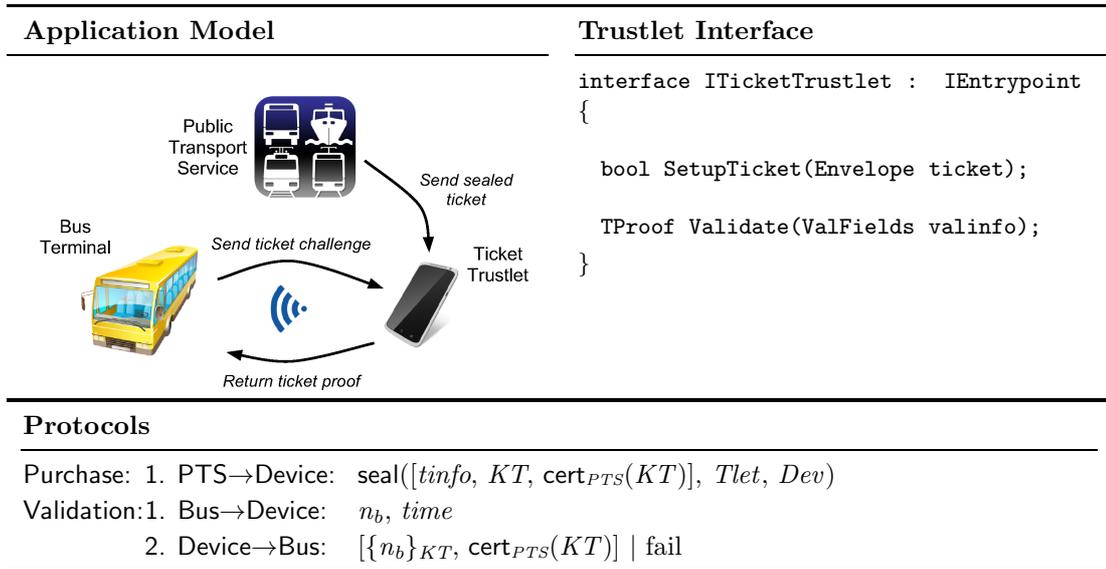


Table 8.2: Use Case 2: Mobile ticketing.

The trustlet code then locates the given index in the TAN list. If the index is valid, the corresponding TAN is returned to the banking service; otherwise a fail message is produced (protocol Query). The trustlet main method is simple and only requires two methods: LoadTanLst, and GetTan. (Figure 8.4 provides some code snippets of this trustlet.)

8.4.2 User Authentication

Our second use case shows how the TLR could be used for user authentication purposes. In many real world scenarios, individuals use authentication tokens (e.g., tickets or cards) in order to gain access to sites or services (e.g., institutions, public transports, amusement parks, museums, etc.). Authentication tokens can consist of physical objects (in plastic or paper) or digital objects (e.g., a barcode or QR code). In addition to replacing physical tokens (just like in the previous use case), the TLR could help improve the security of digital tokens (tickets). In particular, the TLR could provide stronger guarantees against ticket theft. As opposed to existing mechanisms like QR codes that reveal the ticket itself during the ticket validation, the TLR could perform user authentication without the ticket details ever leaving the mobile device. Provided that the TLR sends only a proof of ticket possession to the ticket validator, an attacker has no way to steal the ticket from the device and reuse it in future validations.

Figure 8.2 illustrates how a mobile ticketing application could be built for a public transport company. There are three actors: the *public transport service* (PTS), which issues digital tickets; the *mobile ticket trustlet*, i.e., the trusted code that carries the

digital ticket and produces the ticket proof, and the *ticket validators* (VT), i.e., the ticket readers deployed on the public transport units (e.g., bus, subway).

Essentially, these actors interact twice. When the customer purchases a ticket, the PTS issues a digital ticket in a sealed envelope and sends it to the customer's device, where the mobile ticket trustlet is expected to run and to securely maintain the ticket (protocol *Purchase*). Later, before getting access to public transportation, the user validates the digital ticket by swiping the device in front of the ticket validator; the mobile ticket trustlet issues a ticket proof and the ticket terminal produces visual or audio output according to the result of the validation (protocol *Validation*). We now explain the details of these protocols.

The central goal of these protocols is to produce a ticket proof that does not leak ticket information. To achieve this, the VT can simply request the trustlet to validate the ticket on its behalf and use the trustlet response as a ticket proof; in this way, the ticket never leaves the device. Naturally, in order for the response to be trustworthy, the VT must assess the authenticity of the trustlet. For this purpose, the VT validates the signature sent by the trustlet in message 2 of the *Validation* protocol. This signature is produced in the trustlet with a secret key KT that the PTS generated and enclosed in the sealed envelope of the digital ticket. Since this envelope is sealed to the mobile ticketing trustlet, KT can only be unsealed if the trustlet running on the user's device is authentic.

The challenge then is to convince the VT that the KT key refers to a digital ticket and was issued by the PTS. To address this problem, the PTS includes in the sealed envelope a certificate $\text{cert}_{PTS}(KT)$ of KT 's public key signed by the PTS. The certificate contains information that identify the type of the ticket and the PTS identity. Thus, by attaching certificate $\text{cert}_{PTS}(KT)$ to message 2 of the *Validation* protocol, the VT can validate that the signature has been issued with a ticket key certified by the PTS.

The protocol also needs to mitigate replay attacks, in which an attacker reuses a legitimate ticket proof (possibly issued by another user) in a future validation. To overcome this problem, each ticket proof is bound to the specific request issued by the VT. In particular, the VT identifies a request using a nonce (n_b) that the trustlet must sign, otherwise validation fails.

One last point must be mentioned in order to fully understand the protocols: the trustlet only produces a signed response if the ticket is valid. To allow for the validation of the ticket, the trustlet must know two pieces of information: the ticket expiration date, and the current date. The ticket expiration date is enclosed by the PTS into the sealed envelope (*tinfa* field). The current date is sent by the VT in message 1 of the *Validation* protocol. The trustlet then validates the ticket by comparing both values. If the expiration date constraints are met, then a signature is produced, otherwise a fail message is returned.

To implement these protocols, the mobile ticket trustlet includes two methods: `SetupTicket` and `Validate`. The former takes the sealed envelope containing the digital ticket (i.e., the data items found in message 1 of the *Setup* protocol), unseals it, and keeps the resulting objects in memory. The latter takes the validation arguments

Application Model	Trustlet Interface
<p>The diagram illustrates the application model for mobile payments. It features three main components: a Bank (represented by a piggy bank), a Vending Terminal (represented by a red machine), and a Payments Trustlet (represented by a smartphone). The interactions are as follows: <ul style="list-style-type: none"> The Bank sends "Send sealed credit card info" to the Payments Trustlet. The Payments Trustlet performs a "Make transaction" with the Vending Terminal. The Vending Terminal sends a "Return credit card proof" back to the Payments Trustlet. The Payments Trustlet sends a "Commit transaction" to the Bank. </p>	<pre> interface IPayTrustlet : IEndpoint { bool SetupCCInfo(SealedCCInfo ccinfo); Nonce InitPayment(); CCProof Pay(TxFields txinfo); } </pre>
Protocols	
Setup:	1. Bank→Device: $\text{seal}([CC, PIN, KT, KT_{ID}, \text{cert}_B(KT), \text{cert}_M(VT)], T_{let}, Dev)$
Pay:	1. Device→VT: n_d 2. VT→Device: $VT_{ID}, amount, date, n_d, n_v, h(PIN, n_d, amount), \sigma_{VT}, \text{cert}_M(VT)$ 3. Device→VT: $[VT_{ID}, amount, date, KT_{ID}, n_v, \sigma_{KT}, \text{cert}_B(KT)] \mid \text{abort}$
Commit:	1. VT→Bank: $VT_{ID}, amount, date, n_v, KT_{ID}, \sigma_{KT}$

Table 8.3: Use Case 3: Mobile payments.

contained in message 1 of the Validation protocol, and returns the ticket proof consisting of the fields enclosed in message 2 of the same protocol.

8.4.3 Secure Mobile Transactions

In our third use case, we show how to use the TLR to perform secure transactions. Specifically, we want to enable customers to perform payments at *point of sale* (POS) terminals by simply waving the smartphone in front of the POS. A POS could be deployed in various contexts: retail shopping, vending machines, toll booths, parking meters, etc. The mobile device keeps track of the customer's credit card details, and engages in a payment protocol with the POS over wireless communication (e.g., NFC). No physical currency like credit card, debit card, or cash would be required. The role of the TLR is then to provide for the security of both (i) the credit card information stored on the phone, and (ii) the payment protocols executed between the device and a POS.

Figure 8.3 illustrates a possible mobile payment scenario involving three actors: the *bank*, which issues credit card information, the *mobile payment trustlet*, which keeps track of the credit card details, and the *vending terminal* (VT), i.e., a POS.

These actors engage in three protocols. The Setup protocol takes place between the bank and the trustlet, and ships the credit card information securely onto the phone. The Pay protocol occurs between the trustlet and a VT during a payment transaction. Lastly, the Commit protocol takes place between a VT and the bank (possibly at a

deferred point in time) in order to credit the payment amount into the vendor’s account. We now explain these protocols in detail.

One way to look at the requirements of the protocols is from the VT’s perspective. The VT must guarantee that the credit card information contained in the customer’s device is valid, and then generate a transaction record that proves to the bank that the payment was performed. To avoid leaking credit card details, the VT tells the mobile payment trustlet to endorse the credit card information and produce a transaction record. (This approach is similar to what is done in the mobile ticketing use case.) The VT then checks the authenticity of the trustlet based on a signature (σ_{KT} in message 3 of the **Pay** protocol). This signature is issued by the trustlet with a private key KT certified by the bank (certificate $\text{cert}_B(KT)$) and enclosed in an envelope sealed to the mobile payment trustlet. The trustlet only issues the signature after receiving the payment details from the VT (payment amount and current date) and validating the credit card expiration date. This signature is sent along with $\text{cert}_B(KT)$ in message 3 to VT. By validating the signature against the certificate, the VT can check the authenticity of the trustlet. Message 3 serves as a transaction record that can be forwarded to the bank. (To prevent replay attacks, a nonce n_v sent by VT to the device must be included in the signature.)

From the customers’ perspective, it is important to prevent the impersonation of legitimate terminals. To authenticate a VT, the trustlet verifies whether the VT owns a private key that has been certified by a trusted VT manufacturer. To enable this, the bank includes in the sealed envelope certificates of trusted VT manufacturers (field $\text{cert}_M(VT)$ of the **Setup** protocol). Then, in the **Pay** protocol, the trustlet does not issue a payment signature unless it receives an authentication proof from the terminal. Such a proof consists of issuing a signature σ_{VT} (with the VT private key) of a trustlet chosen nonce n_d . The VT sends the signature and its certificate $\text{cert}_M(VT)$ so that the trustlet can validate them against the certificates of trusted VT manufacturers.

From the customers’ perspective, in addition to making sure the terminal is trusted, we must prevent accidental or abusive payments in trusted terminals (e.g., duplicate payments). For this reason, customers must authorize the payments by typing a PIN, which must be validated by the trustlet before authorizing the payment. Since the I/O path between the mobile device’s UI and the trustlet can be intercepted by a possibly compromised OS, the PIN must be typed in the VT. The VT hashes the PIN along with nonce n_d and payment amount, and sends the result to the trustlet as authorization proof. Sending this hash value (i) prevents the PIN from being sent as clear text, and (ii) binds the PIN to that particular transaction, thereby preventing replay attacks. The trustlet validates the authorization proof by recreating the hash and comparing them. To recreate the hash, the trustlet finds the PIN enclosed in the sealed envelope sent by the bank. The PIN is chosen by the customer and conveyed to the banking services before the **Setup** protocol takes place. To modify the PIN, a new sealed envelope must be generated and sent to the device, an operation that could be done on the bank’s website.

To implement these protocols, the mobile payment trustlet implements three methods: **SetupCCInfo**, **InitPayment**, and **Pay**. The first takes the sealed envelope (i.e., the contents of message 1 of the **Setup** protocol), unseals it, and keeps the resulting objects

Application Model	Trustlet Interface
<p>The diagram illustrates the interaction between three entities: Healthcare Authority (HCA), Healthcare Provider (HP), and E-health Trustlet. The HCA, represented by a shield with a caduceus, sends 'Send sealed health records' to the E-health Trustlet (a smartphone). The HP, represented by a doctor icon, sends 'Request records' to the E-health Trustlet, which then 'Return authorized records' back to the HP.</p>	<pre> public IHealthTrustlet : IEndpoint { bool SetupRecords(SealedRecords recs); Nonce InitQuery(); Recs QueryRecs(RecQuery query); } </pre>
Protocols	
<p>Deploy:</p> <p>Query:</p>	<ol style="list-style-type: none"> 1. HA→Device: $\text{seal}([R_0, \dots, R_n], \text{ACPOL}, KT, \text{cert}_{HA}(KT)), Tlet, Dev)$ 1. Device→HP: n_d 2. HP→Device: $[R_k^{ID}], n_h, \{n_d\}_{KH}, \text{cert}_{HA}(KH)$ 3. Device→HP: $[\langle R_i \dots R_j \rangle_K, \{K, n_h\}_{KT}, \text{cert}_{HA}(KT)] \mid \text{fail}$

Table 8.4: Use Case 4: E-health application.

in memory. The second method just returns nonce n_d . The third method takes the validation arguments contained in message 2 of the Pay protocol, and either returns the transaction record or aborts if validation fails.

8.4.4 Access Control to Sensitive Data

Our final use case for the TLR concerns access control to security-sensitive data placed on mobile devices. A compelling example can be taken from the context of e-health mobile apps. The idea underlying such apps is to enable smartphones to carry along the clinical history of their users so that health providers like physicians and hospitals can have quick access to patients' health records in the course of patient visits. Due to the security-sensitive nature of this data, such apps are somewhat controversial, since giving health providers unrestricted access to health records could raise serious privacy concerns. The TLR could address such concerns by restricting health providers' privileges to this data. If we assume the existence of a central Health Care Authority (HCA) that defines access control policies for patients' clinical history, the TLR could enforce the access control policies prescribed by the HCA and provide secure access to health record information placed on patients' phones.

Figure 8.4 shows how this could be achieved. Our e-health app involves three parties and two protocols. These parties are: the *HCA*, the *health providers*, and the *e-health trustlet*. Analogously to the use cases mentioned so far, the e-health trustlet is responsible for securing the health records of the user and enforcing HCA access control policies. These three parties participate in two protocols: the Deploy protocol, which takes place

between the HCA and the e-health trustlet, and loads the health records and access control policies to a patient’s phone; and the `Query`, which runs between a health provider and the trustlet on the patient’s phone, and returns the set of records requested by the health provider after validating the latter’s access permissions. These protocols are designed to provide the following security guarantees to patients and to health providers.

From the patient’s perspective, we want to ensure that only properly authorized health providers can retrieve health records, and that this authorization is granted according to the permissions of the health provider expressed in the access control policy. To enforce this behavior, the HCA sends the health records and respective access control policy to a patient’s phone enclosed in a sealed envelope. The envelope guarantees that these data items can only be recovered by the trusted e-health trustlet. Later, during a patient visit, a health provider can issue a query of health record IDs (R_k^{ID}). As expected, the trustlet’s response depends on the outcome of the policy evaluation. However, in order to evaluate the policy, the trustlet must first authenticate the health provider, who is required to sign a challenge—nonce n_p sent in message 1 of the `Query` protocol. This signature is issued with the health provider’s private key KH , and it must be accompanied by certificate $\text{cert}_{HA}(KH)$, in which the HCA certifies KH ’s public key and further information about the health provider. The trustlet then has all elements needed to validate this signature and evaluate the policy. If authorization is denied, a fail message is sent; otherwise, the queried records are encrypted and sent to the health provider. To make sure that only the health provider can read the records, these are encrypted with a symmetric key K that is encrypted with the public key of the health provider.

The health providers also require some guarantees, namely of authenticity of the received health records. For this, the HCA includes a private key KT and a certificate $\text{cert}_{HA}(KT)$ in the sealed envelope, and the trustlet attaches (i) a signature σ_{KT} of the encrypted records, (ii) the certificate $\text{cert}_{HA}(KT)$, and (iii) the public key certificate of the HA to the last message sent to HP. These elements enable the health provider to verify that the message was signed by the HCA and that the KT key could only be accessed in the e-health trustlet, thereby guaranteeing the authenticity of the received records. To prevent message replays, we include the nonce n_h in the signature.

To implement these protocols, the mobile payment trustlet implements three methods: `SetupRecords`, `InitQuery`, and `QueryRecs`. The first method takes the sealed envelope containing the data items found in message 1 of the `Deploy` protocol, unseals it, and keeps the resulting objects in memory. The second method returns nonce n_d . The third method takes the query issued by the health provider (message 2 of the `Query` protocol), and returns the fields enclosed in message 3 of the same protocol.

8.5 Evaluation

We evaluate the TLR in four dimensions: performance, TCB size, programming complexity, and security.

8.5.1 Performance

To gauge the performance of our TLR prototype, we study its impact on the execution time of applications. In particular, we concentrate on two sources of performance overheads that the TLR applications incur when compared to standard .Net applications: (i) the fact that the trustlet code of a TLR application runs on a slower .Net runtime than the remaining application code, whereas in standard .Net applications all their code runs on an efficient .Net runtime, and (ii) the fact that TLR applications invoke new primitives that contribute to increasing their total execution time.

Methodology

To evaluate the performance of trustlet code and TLR primitives, we run multiple experiments based on micro-benchmarks.

To study the performance of the TLR runtime when executing trustlet code, we used our use case implementation and an additional benchmark suite. The use case prototypes allow us to measure the performance of the TLR for realistic applications. These tests consist of the trustlet code that implements the protocols of the use cases presented in Section 8.4. In total, these trustlets comprise 14 methods: three for Use Case 1 (online banking transfers), three for Use Case 2 (mobile ticketing), four for Use Case 3 (mobile payments), and four for Use Case 4 (e-health application). In addition to the use case prototypes, to better understand the source of inefficiencies of the TLR runtime, we implemented an additional benchmark suite. Since the trustlet code is not allowed to perform I/O operations and is primarily going to perform CPU intensive applications, this benchmark suite consists of 5 CPU-intensive programs: *MatrixMult*, which is a straightforward $O(n^3)$ matrix multiplication program; *Poly*, which computes the value of a 100-degree polynomial using floating point math; *Sudoku*, which is a sudoku solver; *CryptoRSA*, which performs RSA cryptographic operations (signatures, encryptions, and decryptions) using 1024-bit keys; and *CryptoAES*, which performs AES cryptographic operations (encryptions and decryptions) with 256-bit keys. To compare the performance overheads of both the use case prototypes and the benchmark suite, we measure their execution times under two configurations: on the TLR and on Mono.

To measure the performance of the TLR primitives, we implemented a benchmark suite that stresses each of the five operations related to the trustbox lifecycle: *trustbox creation*, *trustlet method invocation*, *data seal*, *data unseal*, and *trustbox deletion*. Since the execution time of some of these operations changes with the size of their parameters, we further implemented some micro-benchmarks for studying this variation. These benchmark programs measure the effect of the factors that are responsible for such variation, namely the cross world communication (relevant in trustbox creation and trustlet method invocation) and cryptographic operations (relevant in seal and unseal).

We run our experiments in the hardware testbed described in Section 8.3. In all our measurements, we run 10 trials and report the mean time and standard deviation.

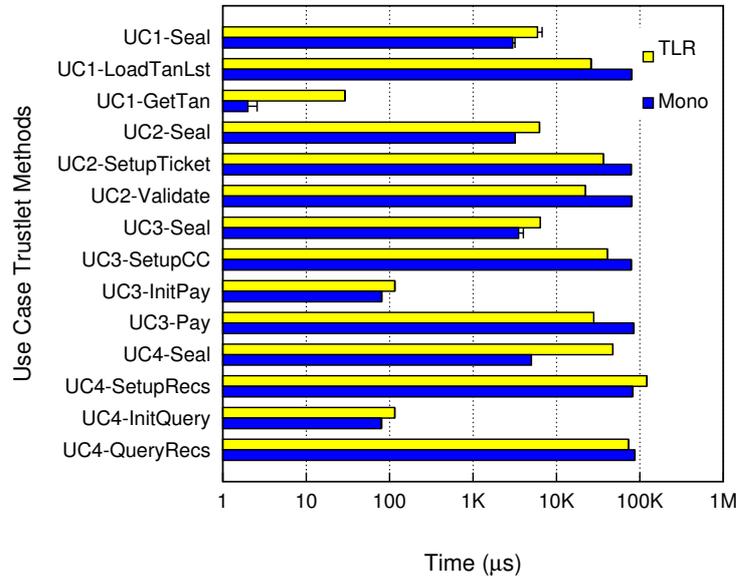


Figure 8.10: Execution time of trustlet methods from our use case prototypes.

Performance of Trustlet Code Execution

Figure 8.10 presents the evaluation results of our use case prototypes. For each use case trustlet, the figure shows the execution time of the trustlet’s methods when executed both on the TLR and on Mono. We can see that the Mono slightly outperforms the TLR: in 57% of the cases, methods execute on average $4.27\times$ faster in Mono than in the TLR; and in 43% of the cases, methods execute on average $2.34\times$ faster in the TLR than in Mono. To a certain extent, these findings were surprising, because we expected Mono to significantly outperform the TLR. This expectation is justified by the fact that, in Mono, the trustlets’ managed code is pre-compiled by a built-in jitter into native code, which runs on bare metal. In contrast, in the TLR, all the managed code is interpreted by the TLR, with the exception of certain libraries, such as the cryptographic library, which are implemented in native code.

To understand why the difference in performance between Mono and the TLR is not more pronounced, we conducted several experiments using our benchmark suite. Figure 8.11 presents the results of our benchmark suite evaluation. As we can see, with the exception of *CryptoRSA*, all other programs of the benchmark run on average $54\times$ slower on the TLR than on Mono. This difference is particularly large in CPU-intensive programs whose managed code the TLR must entirely interpret, such as in the *Sudoku* program, where the difference in performance reaches a factor of 176. However, this difference is clearly inverted in the *CryptoRSA* program, which runs $3.3\times$ faster in the TLR than in Mono, and therefore suggests that Mono’s implementation of RSA is particularly inefficient.

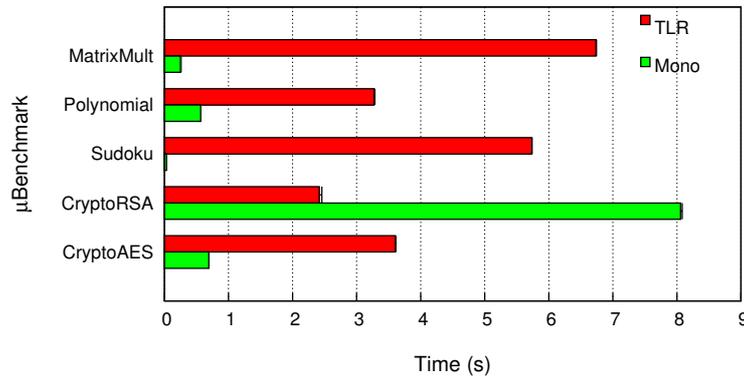


Figure 8.11: Performance of our benchmark suite executed on the TLR and on Mono.

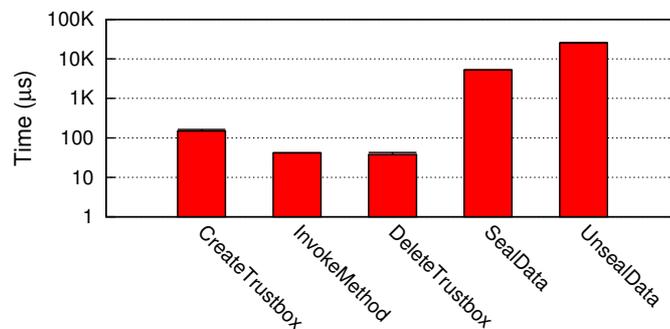


Figure 8.12: Minimum execution time of TLR primitives.

Coming back to the use case evaluation results (see Figure 8.10), we can then understand that the TLR outperforms Mono for the cases where the trustlet code makes more heavy use of RSA operations (e.g., *UC1-Seal*). Mono’s inefficiency, however, is not fundamental, and a performance degradation should be expected for the trustlet code running on the TLR as opposed to running it on a standard .Net environment. Nevertheless, for the realistic use cases we have tested, the trustlet execution time ranged from $29\mu\text{s}$ to 120ms, numbers that did not negatively hurt the user experience.

Performance of the TLR Primitives

To evaluate the performance of the TLR primitives, we measured their baseline execution time, and studied how the execution time of these primitives depended on their input parameters.

Figure 8.12 presents the results of the benchmark suite that measures the baseline execution time of the TLR primitives. While the seal and unseal primitives take on

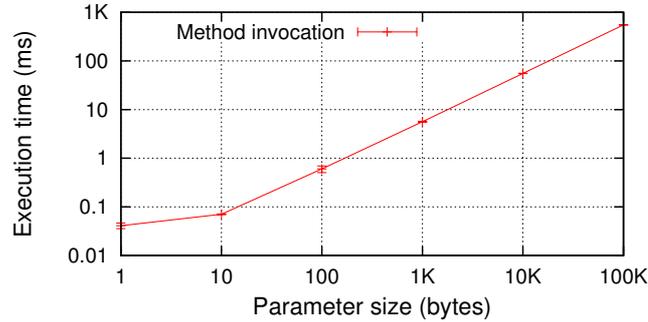


Figure 8.13: Performance of cross world method invocation varying the size of the method parameters.

average 15.2ms, the remaining primitives execute on average in $75.8\mu\text{s}$. This difference is explained by the heavy use of cryptographic operations by seal and unseal.

With the exception of the delete trustbox, which executes in a constant time of $38\mu\text{s}$, the execution time of the TLR primitives depends on their input parameters, namely 1) the amount of data that needs to be transferred across worlds during the trustbox creation and trustlet method invocation, or 2) the amount of data that needs to be encrypted or decrypted by the seal and unseal primitives.

To better understand the cost of cross-world communication, Figure 8.13 plots the execution time of our method invocation benchmark while varying the size of the parameters to be transferred between worlds. The total execution time increases linearly at an approximate rate of $5.6\text{ms}/\text{KB}$. This overhead is explained by the fact that, since the TLR internal data structures, inherited from NetMF, are incompatible with Mono’s, the parameters cannot be transferred across worlds by reference, which would take a constant time. Instead, the parameters need to be marshaled and passed by value, which are operations whose execution time varies with the parameter size. A similar variation could be observed for the trustbox creation primitive, which requires the transfer of trustlet binaries to the TLR in order to instantiate a trustlet object in the trustbox.

Finally, to shed some light on the performance impact of cryptographic operations in the TLR primitives, Figure 8.14 shows our evaluation results for seal and unseal as we vary the size of the data to be sealed and the size of the envelope to be unsealed. Because the TLR makes use of the OpenSSL library to implement cryptographic operations in native code, seal and unseal are efficient. Sealing 1KB takes 5.3ms and unsealing the same amount of data takes 33.6ms. The performance curves of the seal and unseal are dominated by the time complexity of the RSA algorithm, which is used in the implementation of seal and unseal.

8.5.2 TCB Size

To evaluate the TCB size reduction achieved by the TLR, we compare the TCB size of the TLR against that of two representative systems: TrustVisor (see Section 5.4.2), and

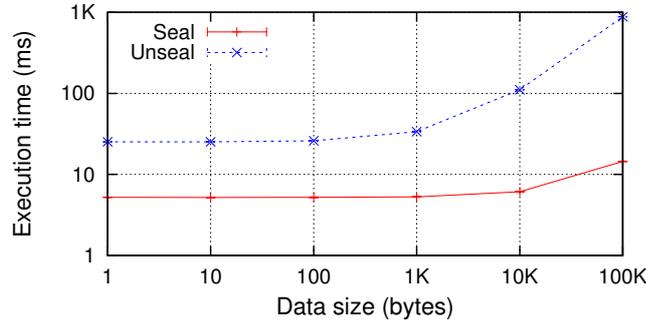


Figure 8.14: Performance of seal and unseal primitives varying the size of sealed and unsealed data, respectively.

Codebase (LOC)		TrustVisor	TLR	Mono+Linux
Managed Code	Libraries	N/A	19.9K (C#)	3,305.3K (C#)
	System	7.2K (C)	52.3K (C++)	7,302.9K (C)
Native Code	Libraries	18.1K (C)	80.5K (C++)	1,308.6K (C)
	System	7.2K (C)	52.3K (C++)	7,302.9K (C)
Total		25.3K	152.7K	11,916.8K

Table 8.5: TCB size of the TLR, TrustVisor, and Mono+Linux setup.

a setup consisting of Mono and Linux (Mono+Linux). While the former gives us an idea of the minimum TCB size achieved by a state-of-the-art system for hosting native code applications, the latter gives us an idea of the TCB size that is currently required for running managed code .Net applications. Since the goal of the TLR is to allow for the execution of managed code applications with a small TCB, comparing the TLR against the TrustVisor and Mono+Linux gives us a good measure of success.

Table 8.5 presents a comparative analysis of the TCB sizes of the TLR, TrustVisor, and Mono+Linux. To measure the size of a system’s TCB, we use the metric of *lines of code* (LOC), which counts all lines of the system’s codebase (including comments and empty lines). In terms of the code versions we considered, for TrustVisor we analyzed version 0.2, and for Linux+Mono we studied version 3.5.1 of Linux and 2.6.7 of Mono. The table also indicates which part of the codebase is implemented in native code (typically in C or C++), and which part corresponds to managed code (typically C#). In addition, the table indicates which part of the code belongs to the core of the system versus libraries.

Comparing the TLR with TrustVisor, we can see that the TLR is approximately five times larger than TrustVisor: the TCB size of the TLR and TrustVisor are, respectively, 152.7 KLOC and 25.3 KLOC. This difference can be explained by the fact that TrustVisor provides hosting capability not for managed code applications, but for native code applications. Therefore, unlike the TLR, TrustVisor neither needs to implement a managed code runtime engine nor to include managed code libraries containing the basic services required by the applications. For this reason, TrustVisor’s core is very small (7.2 KLOC) and includes only a basic cryptographic library (18.1 KLOC). In contrast,

Use Case	Code Size (LOC)	# Methods
Online banking	179	3
Mobile ticketing	450	3
Mobile payments	754	4
E-health app	974	4

Table 8.6: Programming complexity of the use case prototypes measured in code size and number methods.

the TLR must provide support for the execution of trustlet managed code and therefore requires a larger TCB core (52.3 KLOC for the runtime engine and 80.5 KLOC for native code libraries).

Comparing the TLR with the Mono+Linux setup, a configuration that enables the execution of managed code .Net applications, we see that the TLR achieves a drastic reduction in the TCB size. While the TCB size of Mono+Linux is 11.9 MLOC, the TLR’s is 152.7 KLOC, i.e., approximately 60 times smaller. The TCB of Linux+Mono consists of part of the Linux kernel (6.9 MLOC¹), Mono’s runtime (471 KLOC), native code libraries such as the Glib2 (1.3 MLOC), and managed code libraries shipped with Mono (3.3 MLOC). The TLR cuts down the TCB size due to the TLR’s novel design, which restricts the functionality offered to trustlets, and merges the roles of OS and runtime engine into a compact single system.

In summary, we can say that the TLR fills a gap in the design space currently characterized by a tradeoff between TCB size and functionality. On the one hand, we have systems like TrustVisor that depend on a small TCB but operate at a too low an abstraction layer for mobile application developers. On the other hand, we have systems like Mono that provide high-level runtime engines adequate for mobile applications but depend on very large TCBs. The TLR bridges both extremes by providing a high-level runtime engine with essential functionality and a small TCB.

8.5.3 Programming Complexity

Assessing the complexity of programming applications for the TLR is a difficult task. Therefore, our analysis is primarily based on our experiences building the use case prototypes and benchmark programs. We find that it is relatively easy to program applications for the TLR. Once we sketched the security protocols of the four use cases, programming their respective trustlets was done by a grad student in 3.5 days. Table 8.5 shows the codebase size of each trustlet and the number of methods implemented by each trustlet. These numbers show that the average code size is relatively small, consisting of 590 LOC

¹It is unlikely that a real world Linux deployment includes all the device drivers shipped in the kernel. Therefore, to avoid using an artificially bloated Linux kernel, we conservatively exclude the source code of the device drivers. Device drivers entail nearly two thirds of the overall kernel size, which is 15.5 MLOC. Since, in practice, some drivers must be included in the kernel, the TCB size of the Mono+Linux setup would be larger than the number reported in the table.

in C#, and the trustlet interfaces are simple, consisting of 3 to 4 methods. Although it is likely that implementing real world applications would demand a larger programming effort, we believe that building real-world applications on the TLR will be comparable to building them for standard .Net environments, where programmers can take advantage of rich programming environments, language features, and debugging utilities.

8.5.4 Security Analysis

Finally, we discuss some relevant issues concerning the security of the TLR. An attacker wanting to exploit the attack surface of the TLR would face several difficulties. The attack surface of the TLR comprises the `smc` interface exposed to the OS, and the managed code and library interface exposed to the trustlets. Both the `smc` and the library interface are relatively narrow, which reduces the number of vulnerabilities that can be exploited by an attacker. The managed code interface offers a larger attack surface where an attacker could try to exploit a bug in the NetMF engine by providing carefully crafted bytecode sequences in their trustlet code. Although such an attack is possible, the TCB size of the NetMF engine is sufficiently small to be analyzed, thereby reducing the chance of vulnerabilities in the NetMF engine code. Also note that compromising the OS-domain native would not pose any specific problem to the application (e.g., the OS starts forwarding results of a trustlet to a different application). In fact, attacks resulting from an OS compromise can be considered as a particular kind of a man-in-the-middle attack mounted between the trustlet and a trusted remote service, an attack that the application developers already have to mitigate when designing their apps.

The TLR can only provide limited protection against physical attacks: an attacker with the ability to tamper with the hardware could disable the TrustZone protections and over take the defense mechanisms enforced by the TLR. However, such attacks require some degree of sophistication: since the core of the system (the SoC) is packaged in a single die, an attacker would need to break into the SoC in order to conduct this attack successfully. Although this task is not impossible, it would be extremely difficult. To prevent simpler hardware attacks such as probing the bus, the SoC manufacturer could include the memory modules allocated to the secure world directly in the SoC. In this way, bus probing attacks would be impossible without tampering with the SoC.

8.6 Summary

We presented the Trusted Language Runtime (TLR), a system for running trusted applications on the smartphone. TLR offers a trustbox primitive, which is a runtime environment that offers code and data integrity and confidentiality. With the TLR, programmers can write managed-code applications in .Net and specify which parts of the application should run inside a trustbox. These parts, called trustlets, are protected from the remaining code running on the smartphone, including its OS and other applications.

TLR uses the ARM TrustZone, which is a hardware technology for trustworthy computing found in ARM chips. The rich hardware support offered by ARM TrustZone combined with the flexibility of the .Net programming environments allows the TLR to

8 *Trusted Language Runtime: Enabling Trusted Applications on Smartphones*

offer a secure, yet rich programming environment for developing trusted mobile applications. In addition to presenting the design and a TLR implementation based on the NetMF, this chapter showed that the system performs well, and that it can successfully host an array of compelling applications with stringent security needs.

9 Analysis and Limitations

In this chapter, we aim to show how the systems presented in this thesis fit together and get a deeper understanding of their limitations. To this end, we start by providing a unified design model of these systems based on trusted computing—in fact, despite their diversity, they all borrow their core design principles from trusted computing. Then, we leverage this model to discuss the main limitations of these systems (and of trusted computing in general), hoping that this discussion will help identify open research questions to be addressed in the future.

9.1 A Unified Model for Trusted Computing Systems

Essentially, a trusted computing system aims to improve users' trust in a particular computing platform by granting access to users' data on the platform *if and only if* the platform executes a state machine that the users have approved. Since the state machine of a platform defines its expected properties (e.g., confidentiality and integrity protection of computations), by approving a *trusted state machine* that implements certain required properties, users can obtain *a priori* guarantees that those properties will be enforced before uploading their data to the platform.

To restrict data access based on a trusted state machine, trusted computing systems adopt a similar high-level design, whose components are shown in Figure 9.1. The trusted state machine is specified as a piece of *trusted software* (e.g., a hardened hypervisor as described in Section 3.1.3). Given that a general purpose computing platform can boot an arbitrary piece of software (and therefore execute an arbitrary state machine), to validate the software executing on the platform, the system provides a set of *trusted computing primitives*. As introduced in Section 2.4.1, these primitives typically include

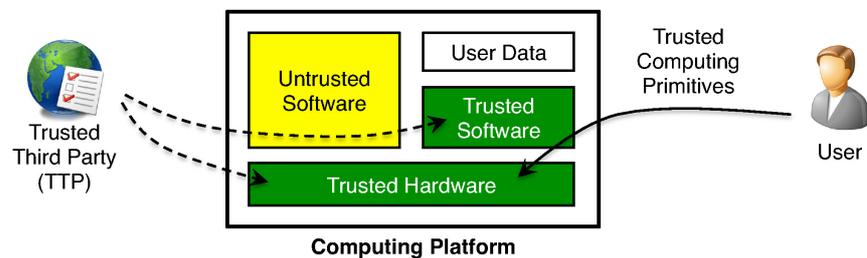


Figure 9.1: Key elements of a general trusted computing system: *trusted software*, *trusted hardware*, *trusted computing primitives*, and *trusted third parties*.

9 Analysis and Limitations

trusted boot, *remote attestation*, and *sealed storage*. To protect the integrity of these primitives, their implementation is grounded on a piece of *trusted hardware* (e.g., a TPM). Since the trusted software and hardware components are not directly controlled by the users, users must ultimately rely on *trusted third parties* to correctly implement and certify these components. For a deeper exposition of some of these concepts, we refer the interested reader to [PMP10].

As described next, this general design can be used to model the components of the trusted computing systems presented in this thesis: Excalibur (for cloud platforms), BROKULOS (for enterprise platforms), and the TLR (for mobile platforms). We clarify how each component is implemented by each system.

Trusted software: The trusted software is typically tailored to enforce specific properties. In Excalibur, the trusted software consists of the monitor code and the client side libraries located on the cloud nodes. These components implement the policy-sealed data abstraction, which can be used for bootstrapping trust in the cloud (see Section 4.2.1). In BROKULOS, the trusted software consists of (i) the trusted programs (brokers) that administrators use to maintain the operating system, and (ii) the software that runs in privileged mode, namely the kernel and OS services. By implementing the security invariants of the Broker Security Model (see Section 6.1.1), the brokers offer the administrators the tools to manage the system without compromising the confidentiality and integrity of users' data and computations. Lastly, in the TLR, the trusted software comprises the TLR code and the security-sensitive app code (trustlets). The TLR code guarantees the protection of confidentiality and integrity of the trustlets' runtime state (see Section 8.1.2).

Trusted computing primitives: To allow for the validation of a platform's configuration, a trusted computing system typically provides trusted computing primitives, each of them serving specific purposes. Excalibur provides an *attest-monitor* primitive for attesting the configuration of the monitor, and *seal* and *unseal* primitives for sealing and unsealing policy-sealed data (see Table 4.4). BROKULOS embeds trusted computing primitives in the implementation of certain brokers, e.g., the broker for activating a user account the first time a user logs in includes an attestation mechanism that checks the integrity of the system (see Section 6.3.3), and the brokers for backing up and restoring user data implement a mechanism akin to sealed storage to guarantee that the backed up data is encrypted before leaving the user's account and can only be decrypted by the user (see Section 6.3.2). Lastly, at the API level, the TLR provides *seal* and *unseal* primitives that enable validating the integrity and identity of a specific trustlet before entrusting a trustlet with sensitive data (see Section 6.3.3). In all these cases, the implementation of the trusted computing primitives is rooted in trusted hardware.

Trusted hardware: To implement the trusted computing primitives, we use two trusted hardware technologies. In the cloud and enterprise setting, Excalibur and BROKULOS, respectively, use TPMs deployed on the local machines. In the mobile platform, the TLR

leverages ARM TrustZone technology. There are, however, other technologies that could be used: some are already available, while others are in the making. An example of an alternative technology currently available is that of the IBM cryptographic coprocessors. The 4765 Cryptographic Coprocessor [IBM13], for instance, is a tamper-resistant PCIe card, containing a complete programmable subsystem (CPU, RAM, persistent memory) and specialized hardware functions (e.g., hardware random number generator, time of day clock, cryptographic functions). The coprocessor can execute security sensitive applications, relieving the main processor from those tasks. The downside of this technology is currently its price, costing over \$9000 per unit. Intel is currently working on a new technology called Software Guard Extensions (SGX) [MAB⁺13]. Essentially, it consists of a set of extensions to Intel processor architecture that enables applications to execute with confidentiality and integrity in the native OS environment. Applications can allocate protected containers called *enclaves*, consisting of a set of protected memory pages inside the application's address space. SGX offer mechanisms for securely loading code and data into the enclave and for encrypting its content in case it needs to be offloaded from main memory. Since SGX is expected to appear in commodity Intel processors, it is likely that the prices will be more competitive than IBM's coprocessors and therefore that SGX will become more widely available.

Trusted third parties: In addition to the manufacturers of the trusted software and hardware components and the Certificate Authorities of Public Key Infrastructure (e.g., VeriSign [Ver]), each system requires specific trusted third parties (TTPs). Excalibur depends on *certifiers* responsible for issuing certificates for the attributes of a particular cloud service (see Section 4.2.1). In BROKULOS, trust is rooted on (a small number of) *fully trusted administrators*, who are responsible for validating the broker implementations and for overriding the broker protections in exceptional occasions, such as for system troubleshooting (see Section 6.1.1). The TLR essentially assumes the existence of application dependent TTPs that trustlets rely on to exchange security-sensitive information, as illustrated by the use cases in Section 8.4.

9.2 Limitations of Trusted Computing Systems

Despite the potential of trusted computing to improve trust in computing platforms, this technology is not perfect. Because trusted computing depends on a few core assumptions to guarantee its effectiveness, some of its limitations are fundamental. Others, instead, are related to its current state of maturity and involve some practical aspects of development and deployment of technology. In this section, we discuss the limitations of trusted computing systems by focusing on each of their components in turn.

9.2.1 Limitations Related to Trusted Software

Since the properties enforced by a trusted computing system depend on a trusted piece of software, two aspects deserve to be highlighted: the need for correctness of the software and the potential vulnerability to side-channel attacks.

First, the trusted software must be correct from both the perspective of its specification and implementation. In fact, if a user attests that a particular computing platform executes trusted software X , but the precise specification of X does not match the user’s high-level idea of what the software is being trusted for (specification problem) or the code does not meet the specified behavior for X , e.g., by containing security vulnerabilities (implementation problem), then it is no longer possible to guarantee that the properties required by the users will be enforced and therefore the attestation result will be meaningless. Producing software that satisfies both these correctness properties is difficult. With respect to the specification, the challenges consist of fully specifying complex pieces of software, and making the specification intelligible to non-technical users. Regarding the implementation, the challenge is to produce provenly correct code for complex and large software stacks. Given the lack of a general solution for these problems, two general approaches have been adopted in order to make the software more robust. One approach is to reduce the likelihood of vulnerabilities in the code by shrinking the size of the trusted computing base (TCB) [MPP⁺08a, MLQ⁺10]. Reducing the TCB size is not easy, as system designers normally face multiple tradeoffs involving, for example, a limitation of functionality, a decrease of performance, and an increase in programming complexity. In a complementary approach the idea is to reduce the attack surface exposed by the TCB by narrowing the system interfaces [MMH08, MSWB09]. Limiting the attack surface reduces the exposure of potential vulnerabilities located in the TCB to an attacker, thereby improving the robustness of the software. To different extents, we have applied both these techniques in Excalibur, BROKULOS, and the TLR.

Caution should be taken to potential side-channels. In fact, as a piece of software executes it could produce *meta-data* accessible to an external observer through side-channels. If this meta-data carries sensitive information, side-channels could be a source of security leaks. One possible side-channel could be the I/O, namely the network. Past research has shown that, even if a network channel is encrypted, the meta-data obtained during the transmission (e.g., IP addresses, size of packets, and transmission time) can be used to compromise certain properties of the communication such as user anonymity [LBMA⁺11] or data confidentiality [CWWZ10]. Another source of meta-data could be the implementation of the software itself. For example, a cryptographic library whose decryption operation depends on the decryption key leaks timing information, which by itself could be leveraged by an attacker to infer the key [BB05]. Yet another source of meta-data could be a shared system resource (e.g., caches, system files, etc.). In certain conditions, a malicious user running a virtual machine (VM) alongside a victim’s VM is able to extract information from the victim’s VM by contending for components of the memory subsystem (e.g., memory pages, and cache lines) [RTSS09, ZJRR12]. Eliminating (or reducing) the meta-data that is leaked through side-channels constitutes an open research topic, and it was not the primary purpose of this thesis.

9.2.2 Limitations Related to Trusted Computing Primitives

Trusted computing primitives serve the purpose of allowing users to “express” trust conditions and enforcing them on a particular platform. In general, trust conditions

regarding the trusted software are expressed as a (hash) function of its binary. For example, sealing data to hash $h(X)$ means that only the software binary X is trusted to unseal the data. However, expressing trust in a specific software implementation raises some obstacles related to the level of abstraction and evolution of the software.

Regarding the level of abstraction, users face a considerable semantic gap between a “hash” and the way they perceive as being the platform’s properties. In fact, users and service providers tend to reason about such properties in abstract terms (e.g., integrity, confidentiality, availability) and express them in a contractual form written in human language, typically in the form of Service Level Agreements (SLAs). With existing trusted computing primitives, however, properties cannot be expressed at this level of abstraction. Because a hash must be provided, it is necessary to build a specific trusted software that implements the properties required by the user. This restriction could be overly inflexible and cumbersome, especially if the service provider operates at upper layers in the software stack (see Section 10.2). To some extent, Excalibur bridges this gap by allowing for the specification of policies based on high-level attributes (e.g., the software version, the location of cloud nodes) and by relying on trusted third parties to map these attributes to the PCRs and AIK keys of TPMs. Nevertheless, there is room for improvement, e.g., by devising primitives that better reflect the concepts and terminology of SLAs, and decoupling them as much as possible from specific implementations.

The tight coupling of hashes with software implementations creates another obstacle as the trusted software evolves and new versions are produced. If a piece of data was sealed to trusted software version X and access should also be given to a future upgrade Y , the previous version X must unseal the data and seal it to Y . (Note that Y could not unseal the data because its hash is different from X ’s.) Implementing this kind of forward portability poses no particular problem and is supported in Excalibur by configuring the monitor with an additional certificate containing a mapping to Y ’s hash). However, revoking the unsealing permissions from a past software version is more problematic. The need for revocation could occur, for example, if a certain implementation is no longer reliable or became obsolete. The systems currently presented in this thesis do not support revocation, and we defer for future work devising a general solution for this problem.

9.2.3 Limitations Related to Trusted Hardware

The fact that trusted computing systems depend on dedicated trusted hardware components raises a few issues that must also be taken into account. Some of these issues are fundamental, as they regard the need for integrity protection of the hardware. Others are specific to particular trusted hardware instances (e.g., TPM and TrustZone), and have to do with their current state of maturity and deployment.

Since trusted computing primitives depend on the trusted hardware, the entire trusted computing system could be compromised if the integrity of the hardware is violated. In other words, trusted computing does not offer protection against physical attacks that result in the leakage of secrets from the hardware or in the modification of the hardware’s behavior. At first sight, this restriction seems to considerably weaken the power

9 Analysis and Limitations

of trusted computing. However, we argue this is not the case for several reasons. Firstly, because the technical evolution of the hardware has significantly raised the bar in terms of the skill, resources, and time that an attacker would require for breaking into the hardware. For example, while early versions of TPM could be defeated by simply interfering with the system bus [Tru08], which is a relatively easy attack, compromising current TPMs requires tampering with the chip itself [Tar10], which is a very sophisticated and lengthy attack. As the degree of miniaturization of integrated circuits increases, attacks of this nature become even harder. Thus, because TrustZone is part of SoC processors and future Intel SGX [MAB⁺13] will be found in Intel processors, physical attacks are very challenging. Secondly, the difficulty of attacks could be made even higher by deploying external barriers to the hardware, such as those located in datacenters (see Section 2.3).

Turning our attention to specific trusted hardware instances, we highlight a few practical limitations related to their deployment. Specifically to TPM technology, an important limitation concerns the certification of TPM cryptographic keys. Two kinds of keys need to be certified. Firstly, the public part of the Endorsement Key (EK) contained in a TPM must be certified by the manufacturer so that the EK can be validated as an authentic TPM key. (The EK is a unique public key pair burned into the TPM by the TPM manufacturer.) Secondly, since remote attestation signatures can only be issued by AIK keys (see Section 2.4.2) and not directly by the EK, the public part of AIK must be certified by a trusted third party in order to vouch for the association between the AIK and a valid EK. Currently, however, there lacks a widely deployed infrastructure to certify these keys. Regarding EKs, most TPM manufacturers do not include certificates to the EK keys of their TPMs. To the best of our knowledge, the only exception is Infineon, whose TPMs contain certificates issued in conjunction with Verisign [Inf05]. Regarding AIKs, Certificate Authorities do not yet offer services for certifying AIK keys. The most popular service used today for this purpose is PrivacyCA [Pri], which however does not offer the necessary security guarantees for a real deployment. The reasons for these limitations are not entirely clear. Nevertheless, they are not fundamental. To overcome these issues, an organization could use Infineon TPMs and delegate the task of AIK certification to an independent department. For security reasons, such a department must not have software administrator privileges over the cloud nodes.

Regarding TrustZone technology, a shortcoming could be its poor availability in commodity mobile devices. Although many ARM-based devices contain processors that implement TrustZone extensions, they do not yet incorporate components that are fundamental for building fully featured trusted computing systems. Examples include a secure ROM, which is responsible for starting the chain of measurements of the trusted boot process, and a unique public key pair, which is fundamental for authenticating the hardware platform during attestation. Just like with TPMs, we do not have an explanation for why hardware manufacturers are not yet taking full advantage of TrustZone. It is our hope that the use cases presented in Section 8.4 help illustrate the benefits of this technology and contribute to changing the state of affairs.

9.2.4 Limitations Related to Trusted Third Parties

The main issues regarding the trusted third parties (TTPs) come from the fact that they constitute the root of trust of any trusted computing system. Trusted computing systems depend on TTPs for a number of crucial tasks, such as correctly implementing trusted hardware and trusted software components, and certifying public keys of all sorts (EK and AIK of TPMs, public RSA keys of organizations and individuals). Therefore, failing to perform these tasks could seriously compromise a trusted computing system. Such a failure could happen by negligence, accidents (e.g., a natural disaster), external agents (e.g., coercion by governmental agencies, security exploits by hackers), or dishonesty (e.g., motivated by a situation of bankruptcy). Independently of the cause, preventing the negative effects of a TTP failure is very difficult. A typical strategy is to spread trust across multiple TTPs, for example, by recruiting multiple TTPs for issuing a certificate or authorizing an operation. This approach, however, is vulnerable to collusion and for certain operations spreading trust across multiple entities is impractical, for example, assembling a trusted hardware component.

The reliance on TTPs is not restricted to trusted computing systems, but to most (if not all) existing systems. Ultimately, the strength of trusted computing systems lies in that (i) it makes the TTPs of a particular computing platform transparent (e.g., Excalibur reveals the hardware and software certifiers), (ii) it can be designed to exclude specific agents from the chain of trust (e.g., BROKULOS excludes partially trusted software administrators from enforcing confidentiality and integrity of computations), and (iii) it gives users of computing platforms the ability to make their trust decision about a particular computing platform (e.g., based on the reputation of the TTPs and on how likely are TTPs prone to jurisdictional interference by governments). For these reasons, we argue that trusted computing systems could help improve users' trust in computing platforms.

10 Conclusions and Future Work

In this section, we summarize the main contributions of this work and outline directions for future research.

10.1 Conclusions

In this thesis, we presented multiple systems aimed at reinforcing user trust in computing platforms. For their popularity and impact, we targeted cloud, enterprise, and mobile platforms. We showed that, in spite of the diversity of these systems, a common twofold strategy can be adopted for building user trust: (i) enhance the security of their software to provide confidentiality and integrity of user computations, and (ii) provide tangible hardware-based guarantees that such a software is really deployed. The core principles to implement this strategy were borrowed from trusted computing, but the specific techniques had to be tailored for each platform. This is because each platform has unique characteristics and usage models that create specific challenges.

In the context of cloud platforms, we had to handle with *massive distribution*. Our motivation was to address the customers' fears of security breaches stemming from insider activity, namely by cloud administrators. To address this problem, we proposed a general *trusted cloud service* design, which includes two kinds of extensions to the cloud infrastructure. The first extension is to reinforce the security of the virtualization software so as to (i) prevent access to in-memory and on-disk customer data by cloud administrators, and (ii) ensure that, as the data migrates across cloud nodes, customer data cannot be inspected or modified on transit. The second extension is to install commodity trusted computing hardware—TPM chips—on the cloud nodes, and leverage TPMs to let users remotely attest the software stack of the cloud, and therefore check that their data is safe. However, we found that, because TPMs have not been specifically devised to large-scale cloud clusters, improper TPM usage could introduce scalability bottlenecks, data migration inflexibility, and privacy issues. To overcome these limitations, we presented Excalibur, a system that enables cloud providers to take advantage of TPMs' attestation properties for building trusted cloud services while using a few simple primitives. We implemented a prototype of Excalibur, and integrated it with an open source cloud platform. Our simulations show that the system can scale to clusters of hundreds of thousands of nodes.

In the context of enterprise platforms, we faced issues of *management inflexibility*. Here, we were also motivated by fears of insider threats, but within the realm of organizations. In general, organizations hold critical data on in-house enterprise platforms, and are currently highly dependent on system administrators for properly maintaining them. A major risk comes from the fact that the operating systems that control these

platforms are normally built for a fully trusted system administrator. While this design model allows for maximal management flexibility, it entails security risks. In particular, it makes the system prone to mismanagement actions conducted by a negligent or malicious system administrator. To make enterprises more resilient to threats of this kind while safeguarding management flexibility, we proposed a *hierarchical privilege separation* model. Under this model most of the management tasks can be offloaded to partially-trusted administrators, without undermining the confidentiality and integrity of user data and computations; only a small number of fully-trusted administrators exists for conducting a small number of critical tasks. We demonstrated that this model is viable in commodity OSes by building BROKULOS. BROKULOS is an extended Debian Linux distribution that disables superuser privileges for the partially-trusted administrators, and allows them to manage the system using only a set of trusted programs called *brokers*. With BROKULOS, we showed that, with about 42 brokers, over 80% of the typical tasks could be offloaded to partially-trusted administrators without loss of confidentiality and integrity of user data.

Lastly, in our work on mobile platforms, we had to address the challenges of *TCB inflation*. In this case, we were primarily concerned about the lack of security guarantees of current mobile platforms for hosting emerging security-sensitive applications, such as e-wallet and e-health applications. We addressed this gap by presenting Trusted Language Runtime (TLR), a system that protects the execution of security-sensitive application components (*trustlets*) inside containers called *trustboxes*. Trustboxes preserve the confidentiality and integrity of application runtime state even if the OS is entirely compromised. The TLR design is novel in the sense that applications can be built using languages that generate intermediate code (e.g., Java and .Net) without bloating the size of the trusted computing base (TCB). This is possible by leveraging ARM TrustZone technology for isolation between the OS and trustbox state, and designing a tiny and carefully crafted runtime engine for hosting trustboxes. In our implementation, we built the TLR by customizing the .Net Microframework (NetMF), a tailored made .Net framework for embedded devices. To demonstrate the TLR, we built applications for four real-world use cases. Our evaluation showed that the TLR can reduce the TCB size of the Mono open source .Net implementation by a factor of 60 with a tolerable performance cost.

10.2 Directions for Future Research

By improving security, the systems presented in this thesis contribute to reinforcing users' trust in cloud, enterprise, and mobile platforms. Nevertheless, a number of directions deserve further exploration. Some of these directions are specific to the computing platforms studied in this thesis, and others concern to trusted computing systems in general.

So far, in the context of cloud computing, we focused on building trust in the lower layers of the cloud stack, namely IaaS. The higher layers of the stack, however, require more extensive work, namely PaaS and SaaS. In these layers, a number of questions

remain open. First, it is unclear whether preventing inspection and modification of customer computations by the cloud administrator can be done while keeping the size of the TCB relatively small. The reason is that, as one climbs the cloud stack, more software needs to be trusted. Second, it is yet to be studied whether the attestation mechanisms we proposed for IaaS would scale in the PaaS setting. Compared to IaaS, PaaS platforms exhibit different workload patterns: PaaS platforms tend to allocate customer software components in a larger number of finer grained containers, place the software components in highly distributed configurations, and migrate them more frequently across nodes. Under such a workload, the demand of attestations could increase to the point of producing bottlenecks presently unknown to us. In the future, we plan to address these challenges and investigate the design of a trusted PaaS platform. To handle the TCB bloating issues, we aim to make use of our past experience with the TLR and leverage some of its techniques.

With respect to enterprise platforms, this thesis has primarily focused on thwarting administration threats in OSes. However, enterprise platforms include additional software components that we did not cover and those also require protection. For example, e-mail, databases, wiki, and web services rely on trusted service administrators who can freely control the user data managed by the services. In this case, just like in an OS, a negligent or malicious service administrator could easily inspect or tamper with user data. To prevent such actions, we plan to investigate whether the broker security model could also be applied, by mediating data access in such services via trusted code (brokers). Ideally, we would like to find a general technique that could provide security guarantees equivalent to those of the broker model, without requiring the handmade design of brokers, a time consuming task.

Regarding the mobile platform setting, we look forward to increasing the functionality and security guarantees of the TLR. As of now, application programmers are somewhat limited in terms of the scope of security-sensitive mobile applications that can be built, namely mobile applications that require interaction with the UI and persistent storage of data. Although the TLR offers runtime protection of security-sensitive application code, it does not presently implement trusted I/O and sealed storage abstractions, which would be required by many mobile applications. Because implementing trusted I/O and sealed storage entails the inclusion of device drivers in the TCB, is not trivial to devise such features without inflating the TCB size. Our future goal is, then, to explore new ways to implement these capabilities and thereby broaden the spectrum of mobile applications supported by the TLR.

Lastly, regarding trusted computing systems in general, there are a number of topics that deserve further research. Since these topics have been covered in detail in Section 9.2, we simply summarize them here. Regarding trusted software, more work is required to handle vulnerabilities in the TCB and side-channels. With respect to trusted computing primitives, more attention should be dedicated to raising their level of abstraction and increase their independence from specific software implementations. As for the trusted hardware, it would be important to assess the degree of physical protection offered by the current technology and, if necessary, improve it. Finally, regarding

10 Conclusions and Future Work

trusted third parties (TTPs), studying new ways to reduce the effects of a TTP failure could represent a significant step forward in the field of trusted computing.

Bibliography

- [acs] Advanced Crypto Software Collection. <http://acsc.cs.utexas.edu>.
- [Age01] National Security Agency. Security-Enhanced Linux (SELinux), 2001. <http://www.nsa.gov/selinux>.
- [Amaa] Amazon. AWS Cloud Computing Whitepapers. <http://aws.amazon.com/whitepapers>.
- [Amab] Amazon EC2. <http://aws.amazon.com/ec2>.
- [Amac] Amazon S3. <http://aws.amazon.com/s3>.
- [amad] Amazon struggles to restore lost data to European cloud customers. <http://www.networkworld.com/news/2011/080911-amazon-outage.html>.
- [App] Apparmor application security for linux. <http://www.novell.com/linux/security/apparmor>.
- [arm] Designing with TrustZone – Hardware Requirements. ARM Technical White Paper.
- [ARM09] ARM. ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
- [Azu] Windows Azure Platform. <http://www.microsoft.com/windowsazure>.
- [BB05] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5):701–716, August 2005.
- [BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *Proceedings of CCS*, 2004.
- [BCG⁺06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *Proceedings of USENIX Security Symposium*, 2006.
- [BCQ⁺11] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proceedings of EuroSys*, 2011.

Bibliography

- [BDF⁺03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, 2003.
- [Bea] Beagle Board. <http://beagleboard.org>.
- [Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 1977.
- [Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of CSFW*, 2001.
- [BLP76] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical report, MITRE Corp., 1976.
- [BOB⁺10] Jeffrey Bickford, Ryan O’Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proceedings of HotMobile*, 2010.
- [BS04] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of USENIX Security Symposium*, 2004.
- [BSW07] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of IEEE Symposium on Security and Privacy*, 2007.
- [CHER10] Cody Cutler, Mike Hibler, Eric Eide, and Robert Ricci. Trusted disk loading in the Emulab network testbed. In *Proceedings of WCSET*, 2010.
- [CLM⁺] Adam Cummings, Todd Lewellen, David McIntire, Andrew P. Moore, and Randall Trzeciak. Insider Threat Study: Illicit Cyber Activity Involving Fraud in the U.S. Financial Services Sector. Technical Report CMU/SEI-2012-SR-004, CMU.
- [CLM⁺07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of SOSP*, 2007.
- [CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of IEEE Symposium on Security and Privacy*, 1987.
- [CWWZ10] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.
- [EGC⁺10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI’10*, 2010.

- [ENI09a] ENISA. Cloud Computing - SME Survey, 2009. <http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-sme-survey>.
- [ENI09b] ENISA. Cloud Computing Risk Assessment, 2009. <http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment>.
- [fbi] Zuckerberg: Facebook 'Made a Bunch of Mistakes' on Privacy. <http://mashable.com/2011/11/29/facebook-ftc-settlement>.
- [Fre] Freescale. <http://http://www.freescale.com>.
- [GBd04] GBdirect. Linux System Administration, 2004. <http://training.gbdirect.co.uk>.
- [Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of STOC*, 2009.
- [Gooa] Google Engine. <http://code.google.com/appengine>.
- [Goob] Google Play. <https://play.google.com/store>.
- [goo10] GCreep: Google Engineer Stalked Teens, Spied on Chats (Updated), 2010. <http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats>.
- [GPC⁺03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of SOSP*, 2003.
- [Gro06] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.
- [gru] Trusted GRUB. <http://trousers.sourceforge.net/grub.html>.
- [Ham07] James Hamilton. An Architecture for Modular Data Centers. In *Proceedings of CIDR*, 2007.
- [HCF04] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation - A Virtual Machine directed approach to Trusted Computing. In *Proceedings of VM*, 2004.
- [HHF⁺05] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. *CollaborateCom*, 2005.
- [hKW00] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of SANE*, 2000.
- [Hyp06] Mikko Hypponen. Malware goes Mobile. *Scientific American*, November 2006.

Bibliography

- [IBM13] IBM. IBM PCIe Cryptographic Coprocessor, 2013. <http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml>.
- [Inf05] Infineon. Infineon Trusted Platform Module Connected to the VeriSign Certificate Infrastructure Chain. Infineon Technical White Paper, 2005. <http://www.infineon.com/dgdl/TPM+Reference+to+Verisign+Certificate+Chain.pdf?folderId=db3a304412b407950112b408e8c90004&fileId=db3a304412b407950112b416601c2053>.
- [ins] Update: Instagram gives in on privacy issues. <http://news.yahoo.com/instagram-policy-changes-post-privacy-challenges-215809883.html>.
- [Int] Intel. Intel Trusted Execution Technology: White Paper. <http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html>.
- [iP02] Security in Plan 9. Russ Cox and Eric Grosse and Rob Pike and Dave Presotto and Sean Quinlan. In *Proceedings of USENIX Security Symposium*, 2002.
- [Jos07] Josep Esteve and Remo Boldrito. *GNU/Linux Advanced Administration*. 2007.
- [JSS] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of SACMAT*.
- [KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of SOSp*, 2009.
- [Kei00] Keith Brown. *Programming Windows Security*. Addison-Wesley Professional, 2000.
- [KMC11] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure data preservers for web services. In *Proceedings of WebApps*, 2011.
- [KYB⁺07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of SOSp*, 2007.
- [KZ10] Taesoo Kim and Nikolai Zeldovich. Making linux protection mechanisms egalitarian with userfs. In *Proceedings of USENIX Security Symposium*, 2010.
- [LBMA⁺11] Stevens Le Blond, Pere Manils, Chaabane Abdelberi, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. One Bad Apple Spoils the Bunch: Exploiting P2P Applications to Trace and Profile Tor Users. In *Proceedings of LEET*, 2011.

- [LSWR12] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software Abstractions for Trusted Sensors. In *Proceedings of Mobisys*, 2012.
- [LTM⁺00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of ASPLOS*, 2000.
- [lxc] lxc Linux Containers. lxc Linux Containers. <http://lxc.sourceforge.net>.
- [MAB⁺13] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of HASP*, 2013.
- [MAF⁺11] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are? secure data capsules for deployable data protection. In *Proceedings of HotOS*, 2011.
- [Mas09] Mashable. T-mobile: All your sidekick data has been lost forever, 2009. <http://mashable.com/2009/10/10/t-mobile-sidekick-data>.
- [MCT] Andrew Moore, Dawn Cappelli, and Randall Trzeciak. The Big Picture of Insider IT Sabotage Across U.S. Critical Infrastructures. Technical Report CMU/SEI-2008-TR-009, CMU.
- [Mic] Microsoft. BitLocker Drive Encryption. <http://www.microsoft.com/whdc/system/platform/hwsecurity/default.aspx>.
- [MJB⁺06] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A System for Distributed Mandatory Access Control. In *Proceedings of ACSAC*, 2006.
- [ML97] Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of SOSR*, 1997.
- [MLQ⁺10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.
- [MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *Proceedings of VEE*, 2008.
- [Mon] Mono. http://www.mono-project.com/Main_Page.
- [MPP⁺08a] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of EuroSys*, 2008.

Bibliography

- [MPP⁺08b] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution. In *Proceedings of ASPLOS*, 2008.
- [MSWB09] Andrew G. Miklas, Stefan Saroiu, Alec Wolman, and Angela Demke Brown. Bunker: a privacy-oriented platform for network tracing. In *Proceedings of NSDI*, 2009.
- [Mye99] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of POPL '99*, 1999.
- [net] .NET Micro Framework. <http://www.microsoft.com/netmf/default.aspx>.
- [net07] Porting the .NET Micro Framework. Microsoft Technical White Paper, 2007. <http://msdn.microsoft.com/en-us/netframework/bb267253.aspx>.
- [net10] Understanding .NET Micro Framework Architecture, 2010. <http://msdn.microsoft.com/en-us/library/cc533001.aspx>.
- [NWG⁺] Daniel Nurmi, Rich Wolski, Chris Grzegorzczak, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems. Technical Report 2008-10, UCSB.
- [Ope] OpenSSL. <http://www.openssl.org>.
- [Pan] Panda Board. <http://pandaboard.org>.
- [PKZ11] Krishna Puttaswamy, Chris Kruegel, and Ben Zhao. Silverline: Toward Data Confidentiality in Storage-Intensive Cloud Applications. In *Proceedings of SoCC*, 2011.
- [PMP10] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.
- [pol] Employees Admit They'd Walk Out With Stolen Data If Fired. http://threatpost.com/en_us/blogs/employees-admit-theyd-walk-out-stolen-data-if-fired-061212.
- [Pri] PrivacyCA. <http://privacyca.com>.
- [PRZB11] Raluca Ada Popa, Catherine Redfield, Nikolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of SOSP*, 2011.
- [RKM⁺12] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of IEEE Symposium on Security and Privacy*, 2012.

- [RRT⁺11] Himanshu Raj, David Robinson, Talha Bin Tariq, Paul England, Stefan Saroiu, and Alec Wolman. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, 2011.
- [RSK⁺10] Indrajit Roy, Srinath T.V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of NSDI*, 2010.
- [RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of CCS*, 2009.
- [SdBR⁺11] Emin Gun Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *Proceedings of SOSP*, 2011.
- [SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of HotCloud*, 2009.
- [SJZvD] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of CCS*.
- [SK10] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of Eurosys*, 2010.
- [SMV⁺10] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *Proceedings of WCCS*, 2010.
- [SPD05] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005.
- [SPvDK04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. *Proceedings of IEEE Symposium on Security and Privacy*, 2004.
- [SRF12] Nuno Santos, Rodrigo Rodrigues, and Bryan Ford. Enhancing the OS against Security Threats in System Administration. In *Proceedings of Middleware*, 2012.
- [SRGS12] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of USENIX Security Symposium*, 2012.
- [SRSW11] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proceedings of HotMobile*, 2011.

Bibliography

- [SRSW13] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted Language Runtime: TCB Reduction for Mobile Applications. Technical report, MPI-SWS, 2013.
- [SS04] Ahmad-Reza Sadeghi and Christian Stübke. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of NSPW*, 2004.
- [SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security Symposium*, 2004.
- [tan] Transaction authentication number. http://www.wikipedia.org/wiki/Transaction_authentication_number.
- [Tar10] Christopher Tarnovsky. Deconstructing a 'Secure' Processor. Black Hat, 2010.
- [Teg] Tegra 250 Development Board. <https://developer.nvidia.com/tegra-250-development-board-features>.
- [tmo] T-mobile and microsoft/danger data loss is bad for the cloud. <http://arstechnica.com/business/news/2009/10/t-mobile-microsoftdanger-data-loss-is-bad-for-the-cloud.ars>.
- [Tro] TrouSerS. <http://trousers.sourceforge.net>.
- [Tru08] Trusted Computing Group. TCG Platform Reset Attack Mitigation Specification, 2008.
- [U-b] U-boot Bootloader. <http://www.denx.de/en/News/WebHome>.
- [ubu] Ubuntu. <http://www.ubuntu.com>.
- [VEK⁺07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Transactions on Computer Systems*, 2007.
- [Ver] VeriSign. VeriSign Authentication Services. <https://www.verisign.com>.
- [Win08] Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proceedings of STC*, 2008.
- [WOSW04] Lars Wirzenius, Joanna Oja, Stephen Stafford, and Alex Weeks. The Linux System Administrator's Guide, 1993-2004. <http://tldp.org/LDP/sag>.
- [ZBWKM06] Nikolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of OSDI*, 2006.

- [ZCCZ11] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of SOSp*, 2011.
- [ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of CCS*, 2012.

