Securing the Use of Sensitive Data on Remote Devices Using a Hardware-Software Architecture

Jeffrey Scott Dwoskin

A Dissertation
Presented to the Faculty of Princeton University in Candidacy for the Degree of Doctor of Philosophy

Recommended for Acceptance by the Department of Electrical Engineering
Adviser: Ruby B. Lee

June 2010
© Copyright by Jeffrey Scott Dwoskin, 2010.
All rights reserved.
Abstract

Many corporations, private organizations, and government agencies maintain sensitive data that must be accessed remotely by their employees using portable devices. The organizations have a responsibility to secure the data to ensure that it does not get used inappropriately or get disseminated beyond these trusted users. We have designed a computer architecture for these devices, combining new hardware and software, that allows trust to be placed in the devices even when they are not under the organization’s physical control.

We have designed, implemented, and tested the Authority-mode Secret-Protection Architecture, which places roots of trust in hardware in the processor chip. It provides new hardware mechanisms based on these roots of trust to protect the execution of trusted software and to provide that software with master secrets. The software uses the master secrets to secure the sensitive data and to communicate securely over the network. The user interacts with this software, which enforces security policies while giving access to data.

The organization designates a central authority that will manage the software on the devices, set security policies, communicate with the devices, and control access to data. Our new hardware mechanisms bind together the device’s on-chip roots of trust with the authority’s data and trusted software, such that the authority can be assured that the security policies will always be enforced.

To show how our design can be adapted to other platforms, we provide a modified architecture for embedded devices. We additionally demonstrate how the full architecture can be integrated with trustworthy system software in a mandatory access control system.

Finally, we have built a testing framework that can help designers validate new security architectures like ours. The framework allows new architectures to be modeled in a virtualization environment, where a separate testing system has complete controllability and observability over hardware and software. It is used to test the effects of various security attacks and to assist in the development of trusted software for the new architecture. We use the framework to test the prototype hardware and software of our architecture.
I would like to thank all of my family, friends, and collaborators who have been a part of my life. You have not only made it possible for me to complete graduate school and my dissertation, but also made it a personally and professionally rewarding part of my life.

My advisor, Ruby Lee, has dedicated enormous energy and countless hours to helping me through every step of my graduate school career. Her advice and encouragement have been unending and invaluable, guiding me to define the direction of my research. Her attention to detail has helped me hone my ideas and improve the quality of my writing, including my dissertation.

Ed Felten has been an invaluable source of inspiration, support, and knowledge, especially with his recent advice on my dissertation. Ed has welcomed me into his group’s discussions and various projects throughout the years, and has taught me to love computer security and tech policy.

Margaret Martonosi, was not only a reader of my dissertation, but also someone who constantly reminded me of the excitement of academia and research — ever since the visit day that helped me choose Princeton.

I thank Niraj Jha and Mung Chiang in advance for serving as examiners for my dissertation defense. Additionally, Mung has been a valued collaborator, and Niraj helped me at the start of graduate school as my academic advisor.

The staff in the Electrical Engineering department have also been invaluable during my time at Princeton. I consider each of them my friends and colleagues; their daily support has kept me grounded and focused during my work. Karen Williams was the graduate coordinator when I applied. She contributed greatly toward my decision to attend Princeton. To this day I remember her instructions to locate the Engineering Quad, describing it as the “ugly red brick building,” which did little to help me identify the building as someone coming from Rutgers. Tamara Thatcher and Sarah Braude have similarly brought their charm and energy to the department and my time here, and have generously applied their serious baking skills, the fruits of which I have thoroughly enjoyed. Stacey Weber has great wit and always has my back. I also greatly appreciate the friendship of these other staff members: Lori Bailey, Meredith Weaver, Anna Gerwel, Sania Sadhvani, Colleen Conrad, Radhika Mohan, Linda Armour, Linda Dreher, Roelie Abdi, and Kathryn Chizik.

The members of my research group have been close collaborators, office-mates, and friends: Patrick McGregor, Zhijie Shi, Xiao Yang, Murat Fiskiran, Zhenghong Wang, Nachiketh Potlapally, Yedidya Hilewitz, David Champagne, Peter Kwan, Yu-Yuan Chen, Jakub Szefer, Sunha Ahn, Reouven Elbaz, Cedric Lauradoux, Mahadevan Gomathisankaran, Michael Wang, and Ray Cheung. I thank Mahadevan especially for his collaboration on the Testing Framework project, and Dahai Xu and Jianwei Huang for their collaborations on the Embedded SP project.

I’ve also had the privilege to work with professors and researchers at other institutions. In particular, I thank my collaborators on the SecureCore project: Cynthia Irvine, Tim Levin, Thuy Nguyen, and Paul Clark at the Naval Postgraduate School and Terry Benzel and Ganesha Bhaskara at the Information Sciences Institute.
My family members are of course the most important people in my life, and their support has been continuous and significant. Loving thanks to my parents: Ellen Fellner and Steven Dwoskin; my step-parents: Alan Fellner and Nanette Dwoskin; my grandparents: Rose and Dave Linden, and Shirley and Harvey Dwoskin; my siblings: Mark Dwoskin, Daniel Dwoskin, Hank Fellner, Rebecca Jarzyna, and Jessica Holloschutz; my step-grandparents: Serri and Fred Barko, and Josephine and Jack Fellner; and all of my aunts, uncles, cousins, nieces, and nephews.

In addition to those mentioned above, my colleagues from internships have contributed to my professional life: Mike Chen, Carl Waldspurger, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyan, Dan Ports, Julia Austin, and others at VMware; Bill Horne, Tomas Sander, Stuart Haber, Jack Brassil, and others at HP Labs; Max Christoff at Morgan Stanley; and Michael Poslusny at Microsoft.

At Princeton, I also would like to recognize the collaborations and friendships of the members of the Center for Information Technology Policy: Ari Feldman, Bill Zeller, Harlan Yu, Shirley Gaw, David Robinson, Laura Cummings-Abdo, Scott Karlin, Joe Calandrinio, Will Clarkson, Tim Lee, Jen King, Josh Kroll, Deven Desai, Jens Grossklags, and Steve Schultze; and other close friends from the Electrical Engineering department: Jiayue He, Shannon Hughes, Chris Sadler, Noel Eisley, Chad and Sasha Myers, Pei Zhang, Vassos Soteriou, James Donald, Eugene Brevdo, Abhinav Agrawal, Julia Chen, Joe Hamburg, Jason Schlessman, Natalie Kostinski, Adam Stoller, Divya Arora, Medini Padmanabhan, Manos Koukoumidis, and Kostas Aisopos. I must single out Jiayue (Jenny) as a close confidant and someone who always impresses me.

I’ve had the pleasure to live with many great roommates in college and graduate school. Eric Burns, despite not owning a nuclear power plant, has been a companion through the endless late night lab reports and the bean-shaped clouds that led us both to pursue our Ph.Ds. Gilberto Contreras studied with me for general exams and then became a very close friend; he is the master of intricate, sometimes confusing, nicknames. “Professor” Alex Halderman will in all likelihood take over the world someday, although with some claim of having won the election; we can only hope that he will be a benevolent dictator, at least to his friends. Katie Mack kicks ass, although she would never say so herself. Melody Chan was almost my roommate, and as one of my best friends she is entitled to the honorary title Regardless of where she lives. Shin-Yi Lin and Matthew “Your Lordship” Weber have been close friends for years and welcomed me into their home and their family. I consider all of you to be family, and couldn’t have asked for better people to invite into my home over the years.

Jillian Lang, Sarah Frischberg, Liz Jobson, and Ma’ayan Bresler have each been very special people in my life who make me happy. Donna Sy and Kolia Sadeghi are also two of my closest friends, travel companions, fruit gatherers, and dance instructors.

I appreciate Andrew Reitano for being the funniest person I’ve met to this day, and Juan Melli, who I cannot imagine being anything other than a “squeaky-clean citizen”, despite his great mind for innocent pranks. Their wives, Karen Reitano and Stephanie Melli, are also great friends and some lucky ladies. Jill “Love Schak” Blumenstock is
another lifelong friend, along with her husband Fred. My other Rutgers friends also
deserve credit for making my college years extraordinarily fun: Gina and Tom Reccia,
Jon and Jackie Chain, Cari August, Howard Salis, Tom Jones, Dan Lee, Steph David,
Dean Chen, and Betsy White.

From starting my own business in high school, I have made lifetime friendships
with Obert Reeves Wood III, Jordan Rice, their families, Linda Whitty, and Austin
Heap. Many of my teachers at High Technology High School and at Rutgers inspired
my academic career.

Some of my closest friends have been made through my work on the Graduate
Student Government: Silvia and Sam Bulow, Kelly Kearney, Kevin Collins, Giri
Parameswaran, Christina Hultholm, Aurelien Fraisse, Brookes Brown, Alex Ntelekos,
Dan Raburn, Nicole Esparza and Gabe Rossman, Chris Bienia, Marina Paul, David
Liao, Richard Chiburis, Hilary Bergsieber Khee-Gan Lee, Andrew Ballinger, Jack
Tinsley, Marina Paul, Praveena Joseph-de Saram and Markus Mittermaier, Raja
Chahal, Karin Sigloch, and Aleks Donev. And of course I have to credit (or maybe
blame) those friends who got me started in the GSG: João Boavida, Lior Silberman,
Leonard Pease, Meredith Safran, Anita Adhitya, and Sinead MacNamara.

Of the rest of my friends, I’d like to start by thanking those that I could al-
ways count on for mischief, including the co-founders of the Princeton Gingerbread
Museum: Matt Meola, Sam Felix Taylor, Jessica Hawthorne, Justin Brown, Neven
Fuckar, Ana Pop, CJ Bell, DJ Sethi, and Rodolfo Ríos.

There are also many other close friends who defy categorization: Kim “win-
ner” Kushner, Keren and Yaron Ayalon, Maria Korolev, Sahar Sharifzadeh, Alicia
Juskewycz, Rebecca Fiebrink, Stephanie Goldfarb, Seth Dorfman, Abe Fetterman,
Christina Locke, Louis Lee, Dani Fong, Owen Helmeyer, Abi Go, Fiona Burnell,
Angela Mendelovici, Asa Rennermalm, Damian Carrieri, Shirley Ho, Karen Wolf-
gang, Jason Vagliano, Peter Turner, Teresa Leonardo, Barclay Satterfield, Nadia
Heninger, Silvana Endler, Sarah Halsey, Rebecca Jones, Rodrigo Casassus Coke, Va-
leria Palanza, Jasmine Cho, Mariana Socal, Piper Harris, Rob Harron, Caroline De-
Hart, and Daniell Rowles.

I have enjoyed working with many administrators on various groups and commit-
tees that have provided much needed distractions from my research. A few people
deserve special thanks: Laurel Harvey, Dean Bill Russel, Sandra Mawhinney, David
Redman, Joy Montero, Lisa Schreyer, Beth McKeown, Tara McCartney, Olivia Mar-\ttel, Jenn Telaar, Kim Jackson, Stephen Friedfeld, Rita Saltz, Steve Sather, and Janet
Dickerson.

I've left off many important friends and family members in order to keep these
acknowledgements shorter than the remainder of my dissertation, so a special thank
you to the rest of the people who have been important in my life throughout my
education.

This work was supported in part by NSF-0430487, Cyber Trust Program — Col-
laborative Research: SecureCore for Trustworthy Commodity Computing and Com-
munications.
To my parents and grandparents, who have supported me throughout my education.

In loving memory of Grandma Shirley and Grandpa Dave.
# Contents

Abstract ................................................................. iii  
Acknowledgements .................................................... iv  
List of Tables .......................................................... xii  
List of Figures ......................................................... xiii  

1 Introduction ......................................................... 1  
  1.1 Motivation .......................................................... 1  
    1.1.1 Securing Data on Portable Devices ....................... 2  
    1.1.2 Trust Model .................................................. 5  
    1.1.3 Usage Scenarios ............................................ 7  
  1.2 Our Approach ................................................... 10  
    1.2.1 Threat Model ............................................... 10  
  1.3 Contributions .................................................. 11  

2 Related Work ....................................................... 14  
  2.1 Hardware Architectures .......................................... 14  
    2.1.1 Secure Processors ......................................... 15  
    2.1.2 Secure Coprocessors ...................................... 17  
  2.2 Secure Kernels, Hypervisors and Virtualization ............. 18  
  2.3 Controlling Access to Data .................................... 21  
  2.4 Summary ........................................................ 25  

3 Authority-mode SP Architecture ................................. 26  
  3.1 Introduction .................................................... 26  
  3.2 Trust Models ..................................................... 27  
  3.3 Threat Model ..................................................... 29  
    3.3.1 Assumptions ............................................... 29  
    3.3.2 Threats ..................................................... 29  
  3.4 Architecture ..................................................... 30  
    3.4.1 Hardware-Rooted Trust ................................... 31  
    3.4.2 Trusted Software Modules ................................ 32  
    3.4.3 Secure Execution Environment ............................ 32  
    3.4.4 Secure Storage for Keys .................................. 35  
    3.4.5 Derived Keys ............................................... 38  
    3.4.6 Remote Attestation and Secure Communications .......... 39  
    3.4.7 Device Initialization ...................................... 41  

viii
5.4.5 Remote Key-management TSM ........................................... 92
5.5 Testing of SP ................................................................. 92
  5.5.1 Testing Example ......................................................... 98
5.6 Related Work ............................................................... 98
5.7 Summary ......................................................................... 99

6 SecureCore Platform ......................................................... 101
  6.1 Introduction ................................................................. 101
  6.2 Background on SecureCore Software Stack ......................... 103
  6.3 Protection of Emergency Data ........................................... 105
    6.3.1 Threat Model for Protection of Emergency Data ............... 105
    6.3.2 Secure Communication Channels ................................. 106
    6.3.3 Emergency State Management ..................................... 109
    6.3.4 Containment of Emergency Data .................................... 111
  6.4 Interactions of Multiple TSMs ........................................... 114
    6.4.1 TSMs in User and Kernel Domains ............................... 115
    6.4.2 Virtualization of SP .................................................. 117
  6.5 Security Analysis .......................................................... 118
    6.5.1 Protection of Emergency Data ...................................... 118
    6.5.2 Multiple TSMs .......................................................... 120
  6.6 Implementation of SecureCore Prototype ............................. 121
    6.6.1 Prototype Architecture .............................................. 121
    6.6.2 Key Management and Secure Storage ............................ 123
    6.6.3 TSM Interface ........................................................ 124
    6.6.4 Lessons Learned ..................................................... 129
  6.7 Summary ......................................................................... 131

7 Virtualization of SP and Dual-mode SP Devices ....................... 133
  7.1 Introduction ................................................................. 133
    7.1.1 User-mode SP Architecture ........................................ 133
  7.2 Virtualization of SP ........................................................ 134
    7.2.1 Virtualization of Concealed Execution Mode .................... 134
    7.2.2 Virtualization of Authority-mode SP .............................. 137
  7.3 A Combined User-mode and Authority-mode SP Device ............. 143
    7.3.1 Protecting the User’s UMK on a Dual-mode SP Device ........ 144
  7.4 Security Analysis .......................................................... 148
    7.4.1 Virtualization of SP .................................................. 148
    7.4.2 Protecting the User’s UMK on a Dual-mode SP Device ........ 149
  7.5 Overview of SP Enhancements .......................................... 151
  7.6 Summary ......................................................................... 154

8 Conclusion ...................................................................... 158
  8.1 Future Work ................................................................. 160
    8.1.1 Trusted Software Modules ......................................... 160
    8.1.2 SP Architecture ....................................................... 162
List of Tables

3.1 New Instructions for Authority-mode SP ......................... 33
3.2 Derived Keys for Storage and Communication .................. 39

4.1 New Embedded SP Instructions ................................. 61

5.1 Example Events and Attacks .................................... 74
5.2 TS Controller API for Attack Scripts .......................... 76
5.3 Mapping Attacks to Testing Framework Modules ................ 77
5.4 TF Event and Attack API for Hardware Components .......... 82
5.5 New SP Instructions for Secure Areas ........................ 87
5.6 Example Attacks on the SP Architecture ...................... 93

6.1 Emergency Management TSM Interface ......................... 126

7.1 New Instructions for Virtualization of Concealed Execution Mode . 136
7.2 Master TSM Interface ............................................ 140
7.3 Instructions for Combined User-mode and Authority-mode SP .......... 146
7.4 Summary of All SP Instructions ................................ 151
7.5 List of Possible SP States for the CEM Mode Register ......... 155
7.6 Summary of Modified Processor Operations for SP .............. 156

A.1 TSM Development Best Practices ............................... 168
B.1 Testing Framework Hypercall Interface ........................ 172
C.1 List of Acronyms .................................................. 179
List of Figures

3.1 Trust Model of Authority and Many SP Devices .................. 28
3.2 Trust Model Expanded to Include Third Parties .................. 28
3.3 Authority-mode SP Hardware Architecture ...................... 31
3.4 Signing TSM Code for Code Integrity Checking .................. 34
3.5 Secure Local Storage Hierarchy ................................... 36
3.6 Directory Node Structure ......................................... 37
3.7 Simple Storage Hierarchy ......................................... 38
3.8 Challenge-response Protocol for Generating Session Keys and Attestation 40

4.1 Reduced Embedded SP Architecture ............................... 57
4.2 Expanded Embedded SP Architecture ............................... 59

5.1 Threats and Attacks on Security Architectures .................. 69
5.2 Testing Framework Design ......................................... 71
5.3 Example Event Monitoring and Attack Sequence ................. 79
5.4 Testing Framework Communication Channels ..................... 80
5.5 Overview of Authority-mode SP Architecture ................... 84
5.6 SP Architecture with Secure Areas ............................... 88
5.7 Example Application and Attack Script for Detecting Leaked Keys 97

6.1 SecureCore Software Architecture ............................... 103
6.2 Trusted Channels in the SecureCore Architecture ............... 108
6.3 Data Flow for Emergency Management ............................ 112
6.4 Architecture for Emergency Management Prototype ............. 122
6.5 Keychain and Secure Storage for Emergency Device .......... 124

7.1 SP State Transition Diagram with User-mode Virtualization .... 137
7.2 Secure Storage for the Master TSM ............................... 142
7.3 Hardware Architecture for Dual-mode SP Device ................. 143
Chapter 1

Introduction

1.1 Motivation

Modern corporations and organizations collect and store vast amounts of data about individuals. This data is often sensitive in nature, such as financial transactions, medical records, human resources databases, receipts for purchases, travel history, telephone call records, or government data. Organizations have a responsibility to store and use this data securely and to prevent accidental dissemination to third parties. This task is relatively simple if the data remains within the organization, stored only on secure systems that limit access to only trusted employees and don’t have connections to any outside networks. The task of securing the transmission and use of the data is far more difficult when data needs to be sent to employees and users over the Internet to their portable devices (e.g., laptops, smart phones), which are used remotely off site. We offer a design for a new hardware-software architecture to build into portable devices that protects sensitive data, controls how the data is disseminated, and provides guarantees about how the data can be accessed on the devices.

Access to sensitive personal data while at remote locations is often critical for the performance of certain jobs. For example, a university admissions officer will need access to student applications when they are on the road visiting schools. A lost or stolen laptop in this situation could expose the records of thousands of applicants, who provided social security numbers, identifying information, academic and disciplinary transcripts, and financial information as part of their application. Encrypting the data in storage on the device is only a partial solution since it must be decrypted during operation. Once decrypted, the data is fully accessible to all software on the device, requiring trust in the applications that are running as well as the operating system. A single piece of spyware installed on the machine could leak the entire database of applications in a matter of minutes once connected to the Internet.

Similarly, remote access is needed for sensitive data that belongs to the organization, but may not pertain to a customer or private individual. For example, trade secrets, internal documents, organizational charts, sales data, source code, engineer-
ing designs, etc. Employees will need access to this data, yet the organization has a business need to prevent its distribution to unauthorized persons.

In other cases, individuals may have personal devices where they want access to their own documents, financial information, and communications in addition to data belonging to other groups such as clubs, volunteer organizations, or employers. However, these individuals want to limit the risk of exposure of their data if a device is lost or compromised. They can purchase services from an organization that will manage their portable device to secure the data.

In general, we expect that significant amounts of data will need to be stored locally on each portable device. Employees or individual users who are using devices remotely will often be traveling or in locations where network access is intermittent or otherwise limited. Having a local copy of data ensures that access is available whenever necessary, without the overhead and delay of downloading it in real time. Having local storage of data can also aid in usability since data can be searched or processed efficiently by software on the device. Sending each query to a remote data center is not practical for many usage scenarios.

Given that sensitive data is stored on each device, securing that data in storage and during use by software becomes of critical importance. Organizations, as owners of data, want as much control as possible over how the data is accessed on a remote device, maintaining at least the controls they would have if the data were accessed directly from the organization, with enforcement of policies on every access that is made. They want employees to have access to the data needed for business purposes but no more, and they want to prevent accidental dissemination of the data to others. Individual users want their personal data to be conveniently accessible on their portable devices, without all of their data, or data that was entrusted to them, being exposed if a device is compromised.

1.1.1 Securing Data on Portable Devices

The starting point to protect data is to encrypt\(^1\) it with a cryptographic key and create a MAC (Message Authentication Code – a keyed cryptographic hash\(^2\)) over the data. This will protect the confidentiality and integrity of the data, providing access to read the data only to people who possess the key and allowing them to verify

\(^1\) Encryption takes data that can be readily read and understood (plaintext) and transforms it into a seemingly random set of bits (ciphertext) from which no meaning can be derived. The transformation is based on a key, composed of a string of truly random bits, that is kept secret after encryption. Anyone who possesses this key can perform decryption, using the key to transform the ciphertext back into the original plaintext. However, someone who does not know the secret key cannot make use of the ciphertext.

\(^2\) A cryptographic hash is a one-way transformation of data of any length that produces a short, fixed-length signature representing that data. Any change in the data will result in a different hash value, and one cannot predict how a change to the data will affect the hash value. Therefore, the hash value of some data can be saved and verified at a later time to ensure that the data has not been changed. When a key is used as part of the hash calculation, only someone who knows the key can produce or verify the hash value, and thus the keyed hash can be stored alongside the data and still be verified by a party who knows the key.
that the data was not modified. Data can then be stored on the device in encrypted form without making it vulnerable. The problem is thereby transformed to be that of where to store the key and how to give access to legitimate users without exposing the data to attack when it is later decrypted.

The simplest approach to protecting the key, and one in common use, is to not store it on the device at all, but instead to derive the key from a passphrase that the user enters when accessing the device. This approach is limited because the device can be attacked after the key has been entered. Significant effort is necessary to write software that is not vulnerable to attacks that might expose the key or the data, and even when this can be done, the key still resides in physical memory where an adversary with physical access to the device can use simple methods (e.g., the cold boot attack [75]) to extract keys. Another limitation to this approach is that it assumes that the key and the data it protects only need to be available when the user has logged in to the system and is actively using the device. This is not the case if the device needs to synchronize with network servers in advance or needs to perform computations or pre-process data before the user logs in. Thus it would be advantageous to have a system where keys can be stored securely without being tied to a particular user. Instead, we propose having keys bound to the device, using new hardware mechanisms to ensure that software can manage the keys safely. Users are then separately authenticated by the software before getting access to data.

With hardware protection of keys and basic encryption of data in storage, we can turn our attention to the software on the device which will decrypt the data and make it available to local users. As this software will use keys and sensitive data directly, any vulnerabilities in the software or its execution environment present a risk of exposing the data to attack. We need to establish a Trusted Computing Base (TCB) [44] — the set of all hardware and software components that must be trusted to protect the data. These components must be thoroughly tested to be sure they are trustworthy. The trusted software components should be protected during operation to prevent attacks that might reduce their effectiveness at enforcing security policies. Other components that are not part of the TCB should be assumed to be subject to compromise; the security of sensitive data should be evaluated accordingly.

We assume a typical software stack on our portable devices. The organization provides a user-level application that controls access to sensitive data. This application manages keys, performs user authentication and access control checks, and displays data to the local user. Other user-level applications can be used on the devices, but should not have access to the sensitive data. An operating system (OS) sits below the applications to manage the hardware. This system software is expected to provide file systems, network connectivity, process isolation between applications, and access to display devices and user input. A hypervisor or virtual machine monitor may also be installed beneath the OS to support multiple virtual machines on the devices. It is expected to isolate each of these machines such that they act independently. Each of these software components on the system is a potential target for compromise by adversaries and therefore a potential source of attacks on sensitive data. With physical access to devices, adversaries could easily modify or replace the software. Of particular concern are small modifications to software that leak sensitive data but
are not noticeable to users, when devices are returned to them without evidence of tampering.

When the devices are in use by legitimate users, we trust that they will be conscientious and not permit physical attacks while they work. However software and network attacks are still of serious concern. Most commodity operating systems are quite vulnerable to attacks. They are easily misconfigured to have more permissive sharing than anticipated. Even when properly configured to isolate processes and files that are meant to be protected, there are almost always privilege escalation attacks by which a user process can obtain access to memory, data, and code of other programs. Additionally, there are a steady stream of new remote vulnerabilities discovered, as a result of programmer error, engineering decisions, and the wide range of drivers and software services offered in most systems. Even specialty operating systems that provide more than average levels of security (e.g., SELinux) are very large and will by the nature of software engineering contain many bugs, some of which will be exploitable. It is also typically the case that the more carefully an operating system is selected and designed for security, the more likely it is to be cumbersome to use and configure, to have less robust features, and to be more costly. Most organizations instead use commodity operating systems. Furthermore, very few machines are isolated from network threats and malware. Even controlled corporate networks have been known to have malware introduced (e.g., through employees opening email attachments or responding to phishing attacks and exposing their login credentials). One of our goals is therefore to reduce the quantity of software that needs to be trusted to protect sensitive data, and in particular not to trust the operating system.

Another consideration for not placing trust in the system software is that it is often unreasonable to assume that the organization can have sufficient control over the software that is installed on a remotely used device. Even well intentioned employees will want to install additional software, both for business and personal use. Whether the device is owned by the employee or the organization, any personal use of the device may introduce additional vulnerabilities. In practice an organization may also want to give employees access to data at home or on their personally-owned devices. For example, when telecommuting or connecting to a corporate network with a VPN (Virtual Private Network), it is simple to secure the network but not the end-user device. Similarly, for employees with smart phones, asking them to carry a second device to permit both corporate and personal use is not always practical or convenient. Thus it is advantageous if a single device can be used with a wide range of software, but still be secure enough for access to the organization’s data.

We will present a design for a new hardware-software architecture that permits certain application software to be trusted by an organization without requiring trust in the full software stack. We use hardware features in the processor to provide an execution environment isolated from the operating system, and then add new primitives to support secure interactions with the organization’s systems and permit use of hardware-protected keys, tied to the device, that can be verified remotely. We design the trusted application software to act on the organization’s behalf to control access to data, using the hardware-protected keys as a root of trust for secure interactions with the user and other software.
1.1.2 Trust Model

In general, we consider situations where an organization owns data that it would like to distribute to trusted users on remote devices, while maintaining control over how and when that data is used and how it is further disseminated. The organization enlists a trusted party, referred to as the authority, to manage these devices and the distribution of its data. The organization trusts the authority and delegates to the authority the ability to make access control decisions based on its policies and implement them. To do this, the authority must establish a remote trust relationship with the remote devices, such that it can verify the security of a device before it entrusts that device with the protected data and can renew that trust periodically when it communicates further.

Moving forward, we will discuss the design of the device in terms of its relationship with the authority, which has complete discretion over the use of data. However, as the data is truly owned by the organization, the organization and the authority will have established formal agreements about how the authority is to act with the data. Since these agreements are legal rather than technical, we do not rely on technology to enforce them and do not discuss them in detail.

We assume a one-to-many relationship between the authority and the devices it manages. While the authority will typically control many remote devices, each device is only bound to a single authority. That authority can in turn provide data and policies from many organizations or third party data providers, each of whom trusts and establishes a relationship with the authority. In some cases an organization may act as its own authority for devices distributed to its employees or customers, but we do not need to assume this for the general case.

The legitimate users of these remote devices are assumed to be trusted by the organization and the authority, and are authorized to access the protected data in at least some limited circumstances. This is a clear distinction from users in a DRM (Digital Rights Management) system who may try to access a single copy of some protected data and then share it with others. While the authority will limit the times when a user is trusted to access data and the scope of their access, our users have professional obligations not to disseminate the data they have received. Furthermore, the data owners are primarily concerned with preventing large scale data disclosures, whereas preventing accidental sharing of single records is important, but is only a secondary concern. Overall, data owners want to enable more flexible access to data for their employees and other trusted parties without exposing their databases to unnecessary risk.

\[^{3}\text{An organization may also act as its own authority, configuring and managing the appropriate trusted systems itself. We allow the organization and the authority to be separate entities for cases where multiple organizations want to place trust in the same device via a single trusted intermediary, or when it is more cost effective for the organization to rely on a trusted party that specializes in the development of secure systems and software.}\]

\[^{4}\text{When the data is owned by an individual user, the authority will manage the device and protect the data according to the policies specified by that user in advance. During operation, the local user has limited access to the data, just as if the data were owned by another organization.}\]
We trust the hardware components of the remote devices to operate as designed and to maintain certain security properties. Therefore we can add new security mechanisms to the hardware that cannot be modified or bypassed. Also, since we do not trust commodity operating systems, our mechanisms must operate at a layer beneath the system software.

Our goal is to protect the confidentiality and integrity of the data and policies provided by the authority. The device must enforce the policies when providing access to data for authorized users. Unauthorized users must not have any access to data.

The authority will further employ a defense-in-depth strategy. Most bulk data will only be available to the components with the highest level of trust — trusted software that is directly protected with new hardware mechanisms that can use the keys protected by hardware. This software can regulate access to data and apply security policies with very high assurance, even under physical attacks. When a local, authorized user requests access to some of the data, it must leave this trusted software to be displayed or processed by other software components on the device. Data is therefore released to this other software, which must also employ security mechanisms of its own. Relative to all of the bulk data that is stored and available to the trusted software, only a small amount of data is ever released in this way.

The security mechanisms employed by the other software on the device then form the second level of trust. They are not protected by hardware directly, but make requests into the highly trusted components to access data and make use of protected keys. These highly trusted components provide a root of trust to the rest of the software that would not otherwise be available, making it more effective. System software sits in this second level, and is necessary for interactions with the local user. This software will also need to be evaluated for security; compromises are more likely in these components, and risk must be assessed accordingly. This risk can be mitigated by attempting to reduce data lifetime in the less secure environment. During normal operation with non-compromised software, data released by the trusted software should be used, displayed, and discarded quickly. Thus if the software is later compromised without the knowledge of the local user, only newly accessed data will be put at risk. On the other hand, if adversaries are able to obtain physical control of a device, they will have no authorized access and the trusted software will not release any new data from inside the trusted boundary.

Using this model, we produce a portable device that can be configured by the trusted authority and then provided for remote use to individuals who are authorized to access data. These users have access to data using the full flexibility of software, while the authority can enforce its access control policies over that data and control how information is disseminated. The device provides strong security guarantees to the data owner, without imposing limitations on the functionality of the device for the user.
1.1.3 Usage Scenarios

There are a wide range of scenarios where data is owned and maintained by a central organization, but needs to be distributed to trusted users on remote devices. Here we present a few such examples.

Medical Practitioners  A medical doctor in private practice serves a local community as a primary care physician. She maintains medical records for the services she renders to her patients. She is affiliated with two nearby hospitals and various specialists where her patients are also treated, and has relationships with HMOs and insurance providers. The state’s medical board has decided to setup a system for electronic medical records for practitioners and medical facilities in the state, hoping to improve the consistency and quality of care while reducing the administrative overhead of its members that previously maintained and distributed paper records. The doctor and her peers in many other practices have chosen to participate.

Practitioners and hospitals will each maintain a database of their patient records, granting access to share the records pertaining to patients treated in other facilities. The medical board is setting up an authority to mediate these accesses to medical records between the various organizations and practitioners. It will also tie in to the systems at the insurance companies to simplify billing procedures and to incorporate treatment records from outside the area. It will link to local pharmacies to provide electronic prescriptions and ensure the accuracy of doctors’ orders. Doctors’ offices will keep local, secured copies of the complete records for each patient that they see, combining data from all sources, and their computer systems will give access as needed. The records will synchronize each night, or upon request.

Doctors can enter new data as they see patients, review test results, issue prescriptions, and bill insurance companies. The hospitals are concerned that a single vulnerability at a remote doctor’s office could expose the private records for thousands of patients. At the same time, doctors need a local and reliable copy of all records in their database that they can access without relying on network connections to a hospital’s servers. Each of the doctors’ staff members needs access to this new system from their existing desktop computers that are also used for other purposes.

Our new architecture can be used in the computers used by the doctors and their staff, to achieve the desired access control policies for the security and privacy of the medical records.

Crisis Response  The emergency management agency in a city oversees crisis response operations across multiple government services, including the police department, fire department, and paramedics. Each first responder is given an emergency information device through which they can access sensitive data during an emergency and stay in contact with the command center. Depending on the type and scope of the emergency, a wide range of different data may be needed by each responder. For example, paramedics will need medical records and emergency contacts for victims they are treating; fire fighters will need building floorplans, occupant/tenant lists, and evacuation plans; police officers could need access to criminal records, customs and
immigration data, transportation passenger lists, etc. Each responder will need quick access to specific information from large databases, data that might be restricted at other times and which must not be disseminated beyond those first responders.

Even on the responder’s device, information should only be disclosed as it is needed and to the extent it is needed. For example, while fire fighters might download the entire list of occupants and personnel files for a building, they may only need to access data for certain floors or may read only information about physical disabilities but not home addresses. Similarly, a paramedic might have access to an entire hospital’s database of electronic medical records, but might be limited to viewing at most fifty records per hour, in case an adversary picks up an active device and tries to extract data quickly. In the midst of a crisis it is also likely that some responders will lose their devices entirely, and while they are not logged in, they may have downloaded large amounts of sensitive data for possible use in the field where network access is intermittent.

The same device will also be used for non-emergency operations by the responders, for access to sensitive data as well as to routine data, such as e-mail and websites.

This scenario will be explored in more detail in Chapter 3.

**Statistical Research** Every ten years, the United States conducts a census of all of its residents, covering a wide range of topics and including the majority of the population. The Census Bureau releases various types of aggregate statistical data to the public, but keeps the individual responses secret for many decades. Similarly, data is collected in many other fields, such as in medicine, relating to the spread of illnesses or the results of clinical trials, and on the Internet with archived search histories from web search engines or video rental and recommendation data from online movie rental services. While much research has been done on anonymization of large data sets, it is quite difficult to do so successfully such that individual records cannot be re-identified, especially when correlated with other publicly available data. When there is a risk of databases being accidentally released with private data, the data is simply not made available. This hinders the ability of academics and other researchers to gain access to important data sets to do statistical research for the public benefit. Research in fields like sociology, medicine, psychology, and political science can be deterred as a result. Other times, data is released anyway with severe consequences when it is found to be less anonymous than originally thought.

A device with our new architecture could permit databases, or subsets of databases, to be made available to researchers who work on computers not under the control of the organization that originally collected the data. The trusted software would have access to raw data or partially processed data and could perform additional statistical analysis at the request of the researcher. Only the results of that analysis would need to be released to other untrusted software or to the researcher. When necessary, the trusted software could limit the number and types of queries made on the data to ensure sufficient privacy. It could even scan the results themselves to filter out records or analyses that reveal information too specific about a small number of individuals.
**Mobile Phone Network**  With the proliferation of smart phones to consumers and business professionals, it is becoming more common to have open platforms on mobile phones. Users want to install their own applications and control the operation of their phones, and corporations want to provide access to business applications. This flexibility requires consumers to control the software environment on the phone, possibly introducing new security risks; yet the network provider needs to maintain the security of their network. The provider has software on each device that controls access to the network, providing data connectivity and the ability to make phone calls. Ideally only trusted applications from the provider should have access to these network features and be able to make phone calls over the telephone network, while a limited packet-interface can be provided to user applications for data connections.

Using our new architecture, the provider can supply its trusted application which runs securely on the device with direct access to the mobile data/phone network. This application has a user interface for making phone calls and provides an API to all other software to send and receive data packets. Certain embedded hardware components, including the radio, could be accessible only to the trusted application. This application can then enforce protocols used on the network and limit settings on the radio, which could otherwise be exploited if provided to untrusted software. As a result, even the system software can be made more flexible under user control since it is not given direct access to the mobile network. Malware and user-controlled software would then not pose a significant threat to the network infrastructure.

**Government Agency**  Various government agencies create and maintain sensitive data about individuals and classified data containing government secrets. Like any large organization, they have employees that need remote access to this data on their portable devices. Examples include tax records used by Internal Revenue Service auditors, health records and statistics used by the Centers for Disease Control and Prevention when responding to pandemic illnesses, layouts and operational procedures for power plants and the power grid in the Department of Homeland Security when used for inspection, no-fly lists and details of airport security mechanisms for employees of the Transportation Security Administration, and even war strategies and troop or equipment deployments from the Pentagon used on the battlefield and on military bases. In any of these situations, and many others, government data will be accessed in the field but must be carefully protected.

It is most efficient for the government to use commercial systems rather than develop their own, in-house; these systems provide hardware and software with more features at a lower cost but come with additional security risks. The agencies have a duty to secure their data using best practices and advanced technology, since the cost of a data breach is higher than simply a loss of profit or a corporation’s reputation.

Accessing classified data on the devices adds additional complications to the usage scenario. Often a single government employee will have a high security clearance, but will need to access data at different classification levels at different times, operating in multiple distinct roles with a single device. The data must be protected while in
use on the device, as in our other usage scenarios, but must also be contained such that data does not flow between different security levels.

Our architecture protects trusted software that can enforce complicated access control policies, especially mandatory policies provided by the authority. It can also be integrated with other software security mechanisms on the device that provide more robust control for information flow, such as separation kernels and verified trustworthy operating systems. In Chapter 6, we demonstrate how this integration would work for one possible software system that enforces controls on information flow using these types of software.

1.2 Our Approach

In our new Authority-mode Secret-Protecting (SP) architecture, we place secrets inside the microprocessor as roots of trust. We then add new processor instructions and hardware security mechanisms to access and use these secrets and to protect the execution of trusted software that is provided by the authority. The trusted software can store sensitive data and policies on the device and use the secrets to protect that data. The user will interact with the trusted software, which will enforce the policies as it provides access to data. The authority can also communicate with the device, to authenticate the trusted software using its knowledge of the secrets and to send new data and policies.

Our SecureCore architecture provides a second layer of protection. This architecture uses a secure separation kernel and hypervisor to setup isolated partitions for multiple virtual machines. The trusted software will execute within one or more of these partitions, each running an operating system that is potentially untrusted. All data that is released to the user within the partition will remain contained within the partition; information cannot flow to other partitions or off the device unless permitted by the policies of the separation kernel, which are also dictated by the authority.

1.2.1 Threat Model

The legitimate users of the remote devices are trusted by the authority, and are given access to protected data on the device according to the authority’s policies. These users will authenticate to the device and to the authority as necessary. They will not intentionally redistribute data that they access, attempt to access data beyond their legitimate needs, access data when they suspect their device to be compromised, or participate in attacks. They will take reasonable care to use software and data responsibly. We therefore do not consider insider threats from malicious behavior by these authorized users.

On the device, the hardware and trusted software are designed to protect the confidentiality and integrity of all data and policies. Only the authority can be permitted to set or change these policies. Data must only be made available to the user in accordance with these policies, and must be kept safe from unauthorized
disclosures or modifications. We do not guarantee availability of the data or the trusted software, and cannot prevent most denial of service attacks.

We assume that the authority maintains its own facilities securely. The authority will have physical access to each device for initialization at its secure facility, referred to as its depot. The authority will be responsible for providing the trusted application software for the devices and for the configuration of those devices. Once a device is then released to a user, we assume that the software on the device is subject to attacks from adversaries with physical access and with access over the network. Users may also install their own software, which we consider untrusted.

We consider operational threats to the remote devices and to the network, but do not address developmental threats. Adversaries will only have the opportunity to attack the system after deployment. We assume that hardware designers, software designers, programmers, and system administrators at the authority have good intentions and try to build devices without vulnerabilities or back doors, with thorough testing and security evaluation. In particular, our new hardware features are assumed to work as designed to protect software, execution state, and data on the device. We also assume that during the initialization process, the authority can properly identify and confirm that a device contains our new hardware components.

Placing these new components in hardware is essential, since hardware behavior is not subject to modification even under physical attacks. Thus the security mechanisms cannot be bypassed by the software nor can they be changed after deployment to introduce vulnerabilities. We assume that any testing circuits or firmware update mechanisms in the hardware are disabled or inaccessible to ensure that the behavior of the hardware cannot be changed after initialization. Even the system software cannot bypass any of the security mechanisms in hardware.

The trusted software is similarly tested and evaluated for security. It is trusted with complete access to secrets and for policy enforcement by the authority and is protected by the hardware. The secure hypervisor, if present, is only trusted to contain data that was already released to the user according to the policies.

Commodity operating systems and all other software components on the device are not trusted. Since we consider adversaries with physical access to the device, attacks on software components can be as simple as modifying software on disk when the device is offline. In this way, system and application software could be modified or replaced arbitrarily. For the trusted software and the secure hypervisor, we must therefore protect the integrity of their code and the confidentiality and integrity of their data when the device is offline. During operation we only consider physical attacks for the trusted software, but not for a hypervisor, as discussed in Chapter 6.

1.3 Contributions

This work examines how a hardware root of trust can support security systems on portable devices. We focus on situations where data must be used on devices not under the direct control of the data owner, but where the owner instead establishes remote trust in the device to protect the data. Our contributions are as follows.
In Chapter 3 we have designed a new hardware-software security architecture, Authority-mode SP, for portable devices that provides remote and transient trust for sensitive data. The architecture allows a central authority to disseminate data to remote devices, which through a combination of hardware mechanisms and trusted software are able to protect the confidentiality and integrity of the data, while enforcing access control policies that determine how the data can be shared and used on the device. We develop a usage scenario for crisis response and emergency management, demonstrating remote trust for managing access to data, multiple complementary methods of revocation of data on the remote devices, the secure initialization of the devices, and the use of data by a local user.

In order to more fully explore the usefulness of the architectural concepts inherent to Authority-mode SP, we have created a compact version for embedded devices, described in Chapter 4. In this architecture, Embedded SP, we have extended and applied the Authority-mode SP mechanisms to limited platforms, such as sensor nodes, which have more restricted hardware capabilities and different threat models from the personal computing devices for which SP was originally designed. We design a reduced hardware architecture that provides similar remote trust capabilities for the embedded platform, and demonstrate how it can be applied to securing key-distribution protocols in mobile ad-hoc sensor networks.

Next, in Chapter 5, we describe a framework for testing new hardware-software security architectures, which we developed and implemented. It can verify the security properties of such an architecture during the design-phase, before real hardware can be built. The platform emulates new security mechanisms that would exist in hardware and software in the real architecture, allowing actual software applications to be designed, developed, and thoroughly tested on the platform. The framework provides an environment where the full state of the system under test is available to be actively observed, modified, and dynamically controlled, allowing attacks to be performed that test and stress the security mechanisms, while observing the effects of these attacks beyond just their success or failure.

We have implemented the framework using a virtualization platform and are using it to test the Authority-mode SP architecture. To do this, we have created an emulation module that implements many of the features of the base SP architecture as well as all of the relevant mechanisms for the Authority-mode features. Thus we are able to write trusted software for Authority-mode SP that runs on the platform and can interact with other applications. We then test a variety of interactions on the platform both with and without attacks taking place.

In Chapter 6, we study how to use Authority-mode SP as part of the SecureCore software-hardware platform, as a means to more thoroughly evaluate the software model of our architecture and its interactions with other trusted and untrusted software. We have integrated Authority-mode SP with a secure hypervisor and trusted operating system to produce a full virtualized platform that can enforce Mandatory Access Control (MAC) policies and control the dissemination and use of sensitive data. We use this platform to implement an emergency device that supports Multi-Level Security (MLS) with data at varying security levels separated into different partitions, while still offering the remote trust and revocation capabilities of Authority-mode SP.
We pay particular attention to ensuring that the combined architecture is safe from covert channels. This emergency device demonstrates a method to protect sensitive data during display to the user with trusted I/O paths, while preventing information flow off of the device in digital form. We also solve the problem of how to integrate SP’s trusted software into both the application and operating system layers of the system.

Finally, Chapter 7 discusses some additional enhancements to the SP architecture with designs for virtualizing the SP architecture and for combining user-mode and authority-mode SP in a single device. We further provide a summary of all SP instructions and states that are needed to implement the combined features of all of the enhancements.

We conclude and suggest future research in Chapter 8.
Chapter 2

Related Work

In this chapter we survey related work regarding a variety of security techniques that have similar goals to our own or similar methodology. We start with hardware architecture techniques that improve security of general purpose computers, considering secure processors and co-processors. These are related to the SP architecture in that our roots of trust are placed in the processor chip and our techniques rely on modifications to the processor architecture. Next we review software separation techniques, covering secure kernels and virtualization techniques. Many of these employ trusted software or setup isolated trust domains in comparable ways to our architecture. Our testing framework and the SecureCore project also employ virtualization and separation kernels. Finally, we discuss various ways of controlling access to data. The primary goal of the SP architecture is to protect the confidentiality and integrity of secrets and data, and to enforce access control policies that apply to that same data.

In our work generally, we use strong cryptography and leverage knowledge of security protocols, but the basis of our work is to protect cryptographic keys from software and hardware attacks. This is often assumed, but not solved, by other security mechanisms.

We also discuss specific related work for the testing framework (Chapter 5) and embedded SP architecture (Chapter 4) in their respective chapters.

2.1 Hardware Architectures

In order to solve the remote trust problem, we rely on new security mechanisms in the hardware architecture. We build upon the SP processor architecture [102, 116] as a secure execution environment, extending the architecture to protect third party secrets, enforce access control, and authenticate the system to remote parties. Much related work also explores how to build security mechanisms into hardware to protect against a wide range of threats, or to provide secure execution environments.
Many security architectures have been developed to protect execution of software on local or remote devices, such as XOM [109], AEGIS [171, 170], and others [20, 67, 96, 186]. None address all of our goals, especially enabling transient trust and secrets bound to policies. Like SP, XOM provides a basic secure execution environment which could be enhanced for remote trust, but it also requires more complicated hardware mechanisms, such as asymmetric-key (public-key) cryptography, to achieve its goal of preventing software piracy and code analysis through encryption of code binaries. AEGIS builds on XOM, addressing security issues and providing additional flexibility, and as a result has similarly complex hardware components. Other architectures like Mondrian memory protection [188] provide protection domains in memory for fine-grained separation within and between applications, but do not protect against the operating system or consider the same range of threats that are necessary for our purposes.

ARM TrustZone [9, 8, 7, 187] is an extensible architecture for an embedded SoC (System-on-Chip) where the processor can execute as either a virtual secure processor or as a virtual non-secure processor at different times, with execution resources isolated to protect the “secure world” from the “non-secure world”. Thus security-sensitive code can run separately from regular code and provide security services as needed. The processor core itself does not appear to provide cryptographic protection for code integrity nor the secure storage that is needed to implement SP-like protections according to our threat model. In order to provide the cryptographic root of trust that we use to implement remote trust relationships, additional trusted components must be added to the system bus on the SoC. ARM’s model for such a secure SoC includes secure boot mechanisms, memory and address space controllers, protected memories, permanent keys, and secure I/O paths through trusted peripherals [10]. Such a design is far more elaborate than we need to implement SP components on a similar SoC, but demonstrates the technical feasibility of our hardware requirements.

Much of the past work on secure execution environments, including the Trusted Platform Module (TPM) [174] discussed below, do not protect remote execution without relying on permanent factory-installed secrets or verifying and trusting the operating system [6], and in fact, the entire software stack. Past proposals also used longer public-private keys and more computationally complex public-key cryptography, to permit attestation to arbitrary third parties. We focus on scenarios where attestation is made to a single authority that has initial access to the device before it is used remotely, allowing us to use symmetric keys and much simpler symmetric-key ciphers in the hardware encryption engine.

Secure processors are often considered for use in embedded systems ranging in capabilities from tiny smart cards and RFID chips to smart phones and consumer electronics. Ravi et al. [142] provide a good overview of the challenges involved, discussing how security requirements factor into the overall design of embedded systems and how various types of attacks on devices affect the design decisions. In particular,
discussions of tamper resistance and countermeasures to physical attacks are relevant to any implementation of our work. Kocher et al. [97] expand on these same topics.

Another method where hardware is able to enhance security is the use of hardware monitoring of program behavior during run time. Systems such as [11, 64, 117, 100, 172] use a combination of hardware and static or dynamic analysis of software to monitor program execution for unintended behavior to prevent security violations. These techniques differ from our use of hardware architecture in that they attempt to prevent the exploitation of unintended vulnerabilities introduced during the implementation of software, while we focus on providing new capabilities and a secure execution environment to known-good trusted software. Our secure execution environment aims to provide confidentiality and integrity against even privileged system software. Most hardware monitoring techniques consider the operating system to be trusted and instead aim to detect or prevent attacks such as code injection or privilege escalation due to buffer overflow, format string, or other vulnerabilities in application code. They focus on protecting solely control-flow of applications and sometimes code integrity, mostly against remote or unprivileged local adversaries, but do not address confidentiality and integrity of data for protected applications.

Since these monitoring techniques introduce protection against developmental threats that we do not consider, they could be used in conjunction with our architecture to protect what we consider ‘untrusted’ software components. Hardware monitoring could also be applied to our trusted software, which already has protections against using data memory as instructions, but could be vulnerable to subtle control flow attacks if subject to memory replay or more powerful physical attacks.

Similarly, software-based monitoring systems (e.g., [37]) might also be incorporated into the design of trusted software for our architecture to provide some self-protection to the code for unintended vulnerabilities. Static analysis and taint-checking [128] techniques can be used to evaluate trusted software before deployment. Other software techniques [95] have been used to dynamically protect software from attacks, but are less applicable to our platform since they focus on protecting unmodified code using emulation or binary translation.

Architectures have also been developed to protect data usage and information flow within applications. InfoShield [160] has goals mostly in line with the above hardware monitoring techniques to prevent attacks during execution, but rather than preventing the exploitation itself, it protects the use of application data as a means to avoid the data leakage that results from remote exploitation of vulnerabilities. Code that generates and accesses sensitive data (e.g., encryption keys) is annotated to indicate to the processor when loads and stores to a protected address are permitted, preventing other code from accessing the data at all. Separate code integrity protection is assumed to be present to enforce these mechanisms, and the design does not consider physical attacks or an insecure or malicious operating system. Therefore this technique could also supplement our secure execution environment.

While the aforementioned system protects trusted applications from outside attacks, other information-flow systems aim to control what an untrusted application is able to do with sensitive data. Many of these systems use programmer annotation and language-based or compile-time static analysis to control the flow of data. They
provide assurance to the programmer of policy enforcement when operating on confi-
dential data, but offer only limited assurance to the user, who cannot be sure what
policies the program will enforce reliably. RIFLE [177] improves on these techniques
to enforce users’ information-flow security policies on applications dynamically at
run-time. It translates compiled code into an “information-flow secure” instruction-
set architecture, which then runs on modified hardware with policies enforced by a
trusted operating system. The system controls the flow of data that is accessed by
the untrusted application, giving users confidence that their confidential data is not
used improperly.

2.1.2 Secure Coprocessors

Secure coprocessors are often used to achieve similar goals to secure processors, but
with threat models expanded to trust more hardware components. Rather than
trusted hardware residing inside the microprocessor or System-On-Chip, the com-
ponents may reside elsewhere on the system board, connected via various buses to
the processor, memory, and I/O devices.

The IBM 4758 [55, 163, 162, 54] has been deployed extensively in high security
scenarios for more than two decades. It is a full, isolated computing device that
connects to the host via a PCI interface, providing cryptographic services. It offers
tamper resistance and response for handling physical attacks. Internally, the 4758
uses a secure boot process to maintain a trustworthy software stack, using one-way
ratcheting of the hardware as each software layer is executed to prevent modification
or access to secrets at lower layers. The flexibility of the software allows the 4758 to
be used for a wide variety of purposes, but it remains a high-cost solution for the most
security-sensitive and high-risk scenarios, rather than a light weight, general purpose
security system.

Similar systems that provide secure crypto-processing for specialized or embedded
use are surveyed in detail by Anderson et al. [4], along with an in-depth discussion of
tamper resistance.

The Trusted Platform Module [174], a more general-purpose and widely deployed
coprocessor, protects code integrity and data, but resides off-chip on the system
board. It therefore has a different threat model and assumptions. It does not handle
physical attacks nor does it provide dynamic verification of code integrity, both of
which are addressed by the SP architecture. Physical attacks are likely for mobile
devices that are easily lost, captured or stolen, but TPM’s threat model considers only
software attacks. TPM provides only static verification of code upon program launch,
and does not defend against dynamic hostile code insertion or modification. Instead,
TPM relies on measuring and verifying the entire software stack upon program launch.
This is unnecessarily limiting since it requires keeping track of many variations of
installed software, and verifying the security of large and complex software, in order
to make even the most basic guarantees of policy enforcement and use of secrets.
Furthermore, we expect lower performance using the TPM for all accesses to protected
data. Critical cryptographic operations occur on the much slower co-processor chip
rather than the processor itself, over a slow bus interface, and require much slower asymmetric-key computations.

While our architecture provides remote trust for usage scenarios with stricter threat models, in contexts where the constraints of TPM are acceptable, remote access systems can be built atop this platform. In one such system by Sailer et al. [150], TPM is used to verify the integrity of software on a client machine before it is permitted to access sensitive data from a corporate VPN, with the goal of extending the network controls of the corporate network to the remote machine. OSLO [92] discusses real vulnerabilities in some widely deployed implementations of TPM, and offers a dynamic secure bootloader based on this TPM hardware.

Intel’s LaGrande Technology [87], now called Trusted Execution Technology (TXT) [86, 89], builds upon TPM with processor support for a protected partition. While LaGrande is potentially more secure than TPM alone, it still shares many of TPM’s shortcomings when considered for our purposes. AMD’s AMD-V [3] (AMD Virtualization — formerly called Pacifica) provides similar hardware virtualization support.

2.2 Secure Kernels, Hypervisors and Virtualization

Virtual machines [182, 168, 25, 14, 39, 69] provide another platform for introducing security mechanisms. On many server platforms and data centers, the physical hardware and the hypervisor can be trusted, and protection is only needed within or between separate virtual machines that run on top of the hypervisor and hardware. Even on desktop computers, virtualization can be used to provide isolated execution environments, usually to protect the host machine from misbehaving software within a virtual machine environment. Security kernels [146] in operating systems can provide similar protection for core services from malicious applications. Related work considers how to build secure hypervisors [149, 151] and security kernels, and how to use security techniques in the host to add protection inside virtual machines.

In our work, we provide hardware-rooted protection domains for trusted application software, but still need to integrate with trustworthy system software to provide a secure user interface and to protect plaintext data after the encrypted data has been legitimately accessed and decrypted. The SecureCore architecture discussed in Chapter 6 provides one such system with very strong protection guarantees for military-grade applications. The related work discussed here provides a range of alternative architectures that would also be well suited to integrate with our authority-mode SP architecture for a variety of usage scenarios. The addition of our hardware roots of trust to any of these architectures can make them more robust against both physical and software attacks, such that they can meet the stricter requirements for remote trust.

One aspect of our threat model is the recognition that commodity operating systems are too large to be a reliable part of the trusted computing base (TCB) of a
secure system. However, complex features are needed to support diverse software requirements and a robust user interface. Many software systems attempt to address this by reducing the size of the trusted OS kernel while keeping most OS services outside of the TCB. Microkernels \[110, 78, 70, 83\] often form the basis of such systems, aiming to produce a minimal trusted and privileged OS kernel.

Secure software systems \[79, 77\] can then be built on top of the microkernel. For example, Nizza \[77\] keeps only security-critical components in the TCB, offers a trusted GUI, provides much stronger isolation for applications, and provides special protection domains for operations on secrets and credentials.

More generally than microkernels, security kernels are designs that aim to push policy enforcement and all systems that affect security into the kernel \[146\], forming the core of the TCB.\(^1\) A specific form of a security kernel is a separation kernel, where the kernel provides strict isolation between multiple security domains, e.g., virtual machines, and then mediates all inter-domain communications according to policy \[147\].

Proxos \[173\] takes a different approach to reducing trust in the OS by segregating the system call interface into a trusted set of calls, handled by a trusted private OS, and an untrusted set handled by an untrusted commodity OS. The former is used for security sensitive operations, while the latter allows a private application to interface with other untrusted applications. A virtual machine monitor provides the platform on which the applications and operating systems run, permitting the private application to run within its private OS, but with access to services from the untrusted OS. Applications include private file systems and a trusted I/O path through the private OS.

SELinux \[114\] is a commonly used operating system based on Linux that supports fine-grained mandatory access control (MAC) policies enforced by the OS. It allows the definition of rule sets and object labels with enforcement performed in the kernel, controlling application behavior. It can support a wide range of MAC policies, beyond simply Multi-Level Security (MLS) \[17\], but requires careful configuration for proper enforcement. Overall, this allows for stricter separation of behavior between applications than traditional discretionary user-centric access control policies can provide.

The principle of least privilege \[153\] can also be applied by security kernels and operating systems to isolate the effects of applications on each other and on overall system security. Each application or OS function runs in effective isolation, being given only the least set of privileges necessary to complete its tasks. Thus if a vulnerability is exploited, damage is contained as much as possible. This principle has been applied to research systems, such as Polaris \[166\] which allows applications to be started with limited privileges in Microsoft Windows to prevent the spread of malware, with particular focus on usability issues. It has also been used to some extent in commodity operating systems themselves \[119\], but often in practice only to the extent of limiting the use of administrator privileges. More extensive access control

\(^1\)In systems like microkernels, some trusted components are moved out of the kernel, but are still part of the TCB.
mechanisms already present in commodity operating systems are often difficult to configure properly [72]. The kernel used in the SecureCore project is designed as a least-privilege separation kernel [105, 103].

The concepts for security and separation kernels also underly the designs of secure virtual machine monitors (VMMs). Traditional VMMs [182, 168, 25, 14, 39, 69] expose a virtual machine identical to the underlying hardware, and focus on performance and compatibility with off-the-shelf and legacy software. Secure VMMs must further ensure that security policies are enforced and must mediate information flow between virtual machines. Isolation kernels like Denali [185] ensure strict separation between virtual machines with private namespaces and small codebases for their hypervisors; they can provide a simplified instruction set and limited virtual devices to guest machines to achieve security and performance at the expense of compatibility. Nonetheless, even traditional VMMs aim to achieve performance isolation to ensure that a malfunctioning or malicious guest virtual machine cannot affect other machines. Terra [65] maintains compatibility but takes isolation guarantees further by providing a trusted VMM; it provides both “open-box” virtual machines for running commodity software and “closed-box” virtual machines that can run high-assurance software much like a closed hardware platform would. The closed-box machines run with cryptographic protection, even from the platform owner, rooted in trusted hardware such as TPM. It can then attest the state of the machine to remote parties and provide a trusted path to the user. Other secure hypervisors, like sHype [149, 151], provide strong isolation but also implement reference monitors in the VMM to allow sharing between machines while enforcing mandatory access control policies between them.

In addition to providing isolation, hypervisors can also provide a root of trust for security mechanisms within virtual machines. Overshadow [31] is an enhancement to the VMware VMM that provides confidentiality and integrity to unmodified applications in virtual machines from compromised or malicious guest operating systems. The VMM provides two views of a protected application’s memory space — encrypted and hashed when accessed by the OS, and plaintext when accessed by the application. It then wraps the applications in a ‘shim’ that intercepts and mediates interrupts and system calls to the OS. This provides similar functionality to that of SP, but without hardware support it cannot defend against physical attacks; the full hypervisor and the physical device must be trusted.

Modified hypervisors are also used for virtual machine introspection [66, 136, 111], where software on the host can observe the software configuration or behavior within a virtual machine. Examples include intrusion detection and virus-scanning from non-vulnerable host systems, preventing execution of malware, and tracing memory or disk accesses.

Our work differs from the related work in this section because we develop an integrated hardware-software system where secure, trusted software depends on hardware

---

Footnote 2: Para-virtualized VMMs like Xen [14] do offer a modified instruction set to virtual machines, usually for performance reasons related to use of I/O devices, but still aim to support commodity operating systems and applications, albeit with minor modifications. They remain distinct from systems like Denali which offer a noticeably reduced set of virtualized hardware features.
support to protect sensitive data and policy enforcement. Software alone cannot pro-
tect from the full range of threats that must be considered for remote trust scenarios,
and hardware alone cannot provide the flexibility needed for policy enforcement nor
access to complex data with rich user interfaces.

2.3 Controlling Access to Data

Many hardware and software systems have been built to enforce access control to data
on computing devices, including many of those discussed in previous sections. The
effectiveness of this enforcement depends greatly on the threat model assumed for
the system. Our work on remote trust assumes generally that the authorized remote
user of a device is trusted by the authority that is granting access to sensitive data.
We are thus able to make assumptions about user behavior that make it practical to
secure the data, even though the hardware and software on the device are not trusted
in all respects, and individuals other than the trusted user may be malicious. Once
given access, the trusted user will not disseminate the data or use it improperly.

There is also significant work in industry and academia regarding digital rights
management (DRM), where the user is not trusted; the goal then is typically to
let the untrusted user view or otherwise access content in a controlled environment
while maintaining containment of that same data to prevent piracy (or other forms
of copyright infringement). On a general purpose computing device, this is widely
recognized as a fundamentally more difficult problem, if not futile, by those with
expertise in the field [155, 73, 164].

In actuality, there is a range of possible security policies [1, 44, 17] for controlling
access to data with differing goals and threat models. Park et al. [135] provide a
taxonomy of security architectures that control dissemination of digital content, with
a focus on confidentiality of data and improper use, as compared to limitations on
mass-distribution of paid content that are the focus of most DRM systems. Most
systems in wide commercial use implement some form of Mandatory Access Control
or Discretionary Access Control policies. Additional policy structures that don’t fit
either the mandatory or discretionary model can also be important. For example,
government and military systems often need originator control [134, 115], which aims
primarily to prevent unauthorized re-dissemination beyond the initial recipients of
data.\(^3\) Alternatively, role-based access control policies [62, 61] assign access rights
based on a user’s current role in an organization, and can therefore adjust automati-
cally as the roles of individual users change.

We now look more closely at general techniques available for controlling access to
data, applicable to any access control policy or threat model. We then discuss work
related specifically to digital rights management.

Capability-based systems [43, 59, 41, 153] are often designed as an alternative
means of enforcing access control, to restrict access to secret data or to memory in
general. The hardware will provide protection domains to processes or code modules

\(^3\)A related project from our research group uses the SP architecture to enforce originator control
policies [33].
and then use the possession of a particular token, a capability, to gain access to each
data resource or object. These capabilities are either cryptographically secure and
unforgeable (e.g., for distributed systems) [123], where any entity that presents the
capability will be given access, or are descriptors [189] or memory references [158] that
are tracked and managed by hardware or a secure operating system. Capabilities often
simplify information sharing for access control systems, since the capability itself can
be shared to grant access to another entity, or kept secret to prevent access. On their
own, capability systems can control access to data on a single machine and are useful
for building more secure operating systems (including implementing least privilege
systems), but do not help us with remote trust. Hardware architecture support for
capabilities often result in very complex architecture and lower performance [35].

More recent systems use similar capability-like techniques to support copy protec-
tion and digital rights management on code and data. MESA [161] defines memory
capsules that protect code and data regions with fine granularity within a process or
application. Like XOM and AEGIS, MESA is designed to provide copy protection
for application code, and as a result offers a secure execution environment that might
be used as another platform to implement our remote trust designs in software, as
a side-effect of already severely locking down the system. It relies on new proces-
sor hardware to enforce its protection, but unlike our work, also requires a secure
operating system kernel to verify and manage its security domains. Thus MESA is
also similar to TPM in that it locks down the entire software stack, starting with a
secure BIOS, in order to provide any protection. Our approach is instead to provide a
small secure execution compartment for protected software and remote trust without
trusting or limiting the rest of the software.

Techniques have also been studied for remote attestation before giving access to
data to a remote device. Kennell and Jamieson [93] determine ‘genuinity’ of processor
hardware by remotely querying a device to see if it is running real or simulated hard-
ware. Later work [157, 94] studies various attacks on this scheme, discussing whether
or not it is possible to determine genuinity from processor side effects. SWATT [156]
uses similar software-only techniques to externally verify the memory contents of em-
bedded devices. Generally, these differ from TPM-style attestation in that they rely
on the physical limitations of computing devices rather than purely cryptographic
techniques or trusted hardware components.

Copy protection and attestation techniques are often applied to address digital
rights management, where a consumer acquires limited rights from a content provider
to access digital content, and the content provider uses technology to enforce the
limitations on the consumer’s computing device. Cryptography is useful for secure
content distribution and for high-integrity binding of policies to content, but is no
longer effective when it comes time for the consumer to access the content. Eventually
the content must be made available in the clear (as plaintext bits without encryp-
tion) in order for it to be rendered for display to the consumer. Thus the problem
reduces to that of key management and of restricting the use and redistribution of
the unprotected, plaintext content [15]. In order to succeed, one must generally make
assumptions about user behavior or the level of technical expertise available to an
attacker. This is problematic since the release of a single copy of the unprotected
data by the most capable attacker can make the data available to all users [23].

When content is cryptographically protected, the fundamental limitation is that all
necessary decryption keys must be made available somehow to each consumer in order
to access the content. If decryption algorithms can be broken or cryptographic keys
can be extracted, then the content can be separated from the protection mechanism.
A popular example is the Content Scramble System (CSS) algorithm used to encrypt
content for movies released on DVD, which was broken in 1999 due to the use of short
40-bit keys and weaknesses in the algorithm [165, 16].

In embedded systems, such as a video game consoles, portable media players, and
mobile phones, customized hardware components can be used to enforce access control
policies. These can be viewed as closed systems that are more difficult to circumvent
since the typical consumer does not have the knowledge or tools to modify hardware
components or to make changes to device firmware. However, in practice some experts
can and typically do reverse engineer the platform and provide instructions or software
for others to break the security mechanisms in the device. Closed systems also suffer
from limitations on interoperability [98], which tend to frustrate users and encourage
them to seek alternatives.

For example, mobile phones are often sold ‘locked’ such that they will only work on
a single carrier’s network. Typically this is done since the carrier subsidizes the initial
phone purchase price, relying on monthly fees to recoup the cost, and does not want
the customer to use the phone on a different network. Consumers are frustrated when
they later want to switch providers and keep their phone, or travel internationally
and temporarily use service from a local carrier. Similarly, many phones have their
software restricted to control the application software that will run, often limited to
carrier-provided services that bring in additional revenue (e.g., paying for ringtones
or making calls via the carrier’s voice network with per-minute charges rather than
using voice-over-IP at data rates). Some consumers in turn find ways to unlock or
‘jailbreak’ each phone model [2, 122, 133], to replace or patch the system software to
remove these limitations; these methods are then shared online with other customers,
often embodied in automated tools.

Makers of closed systems also attempt to control content itself, rather than the
devices. Watermarking techniques have been proposed to control the unauthorized
distribution and use of digital content. In such a system, all devices that can access
the content would contain hardware or software that checks for the watermark and
only permits access to the content if the appropriate license has been acquired by the
consumer. Such a system is difficult to deploy since it requires ubiquitous deploy-
ment to be effective; it is also difficult to implement and evaluate robust watermarking
technologies [139]. When the Secure Digital Music Initiative had developed water-
marking to control the distribution of digital music, researchers were quickly able to
disable and bypass the technology [38]. Fuzzy hashing and audio fingerprinting [74]
attempt to achieve the same goal by analyzing content to generate a signature rather
than embedding the signature within the content, as in watermarking; they are simi-
larly ineffective since the content can simply be modified slightly to bypass detection,
usually with the changes and the resulting noise being imperceptible to the consumer.
While closed systems with special purpose hardware present a challenge to the majority of users who try to bypass restrictions, open, general purpose computing systems give the consumer full control over what software to install, in an environment where making changes is relatively simple. Any user with a moderate amount of technical skill can modify and install open-source operating systems; similarly, closed-source commodity operating systems are easily extended with device drivers, kernel modules, and other software that act with full system privileges. In some systems, even application software is run with administrator privileges. Thus the user can have full control over the software environment to more easily defeat technical limitations placed on application software and content.

The Trusted Computing Group’s efforts with TPM attempt to address this by providing remote attestation and sealed storage, but as discussed previously, this type of security hardware only solves part of the problem. TPM can attest to the code that was loaded at one point in time, but does not determine what might have changed after the attestation is made. Measurements must be taken frequently and signatures maintained for a wide range of system components [152]. Furthermore, the attestation cannot determine if the measured code is actually trustworthy; commodity operating systems that provide only discretionary access control and provide relatively open architectures are not well suited to limiting access to content and enforcing policies effectively [144]. In all likelihood, any truly secure and trustworthy system would behave more like a closed system than an open one. Therefore, remote attestation is mostly only useful to verify that a secure kernel or hypervisor is indeed deployed, and even then, hardware attacks remain viable.

Generally, other hardware-based techniques, such as secure processors and co-processors can be applied to DRM to make the platform behave like a closed-system [109, 175, 65].

Another option for using hardware protection on open devices is to move the key management off of the computer itself. Smartcards use tamper-resistant hardware to store and manage the use of keys [12], allowing portability for the consumer while limiting the consumer’s access to the keys themselves. Nonetheless, the plaintext bits of the content must still be sent back to the computer for display, albeit possibly under the control of the closed software components. More importantly, smartcards have been shown to be quite vulnerable to physical attack [5].

Recent industry efforts, such as HDCP [40], seek to complete a secure ‘trusted’-path for general purpose computers, where display hardware includes components that receive protected content in encrypted form and decrypt it just in time to display. In effect, this is another method to embed a closed system into the otherwise open platform. Such attempts are ineffective when content must also be made available to consumers who do not own compatible devices. In the case of HDCP, the technology is also ineffective because weak and unproven cryptographic techniques were used to reduce cost and have been shown to be vulnerable to attack [60].

To combat some of the limitations of closed platforms, research on DRM systems often focuses on how to provide usable systems for controlled access to content to consumers without interfering with their normal activities. For example, Sadeghi et al. [148] show how such a system can be built on top of TPM hardware and secure
hypervisors, and Popescu et al. [141] and Pestoni et al. [138] build DRM systems for consumer electronics and mobile devices that permit sharing between all of a household’s devices, even if some of them do not have network connectivity. Such designs largely ignore the effects of one-time circumvention or complete containment of content, in the apparent hope that usability improvements will reduce consumer frustration and therefore the majority of consumers will not attempt to bypass the technological limitations or obtain unrestricted copies of content through other channels. Knowing that some devices will be compromised, research also focuses on key distribution methods that allow for tracing of misbehaving nodes and revocation of keys [125].

The same is also true of many commercial software-only DRM technologies commonly in use, which make it difficult for the average user to remove protection of content, but pose little to no challenge to a motivated attacker. For example, Apple’s FairPlay DRM software for digital music has been broken numerous times [180, 132]. Software-only systems are fundamentally vulnerable to unauthorized use of decryption keys on the consumer’s device [178] and to replay attacks on systems that require persistent state [159]. Without hardware assistance, the keys that decrypt restricted content must be available on the device in order for the software to access the content, and the content must then be provided in the clear to the operating system and device drivers. Thus software to circumvent access restrictions or to remove DRM protection is relatively easy to develop. Replay attacks are possible since the entire state of the device can be replayed, unless it is rooted in hardware or synchronized with a remote server. Therefore any usage limits can be reset by rolling back the state of the entire system to an earlier time; this is especially simple to do with virtual machines.

2.4 Summary

Our work on the SP architecture enables new methodologies for the enforcement of access control policies, using new processor hardware features and the integration of trusted software into multiple layers of the software stack. We build upon existing techniques for creating secure execution environments using processor hardware and virtualization, but define new hardware roots of trust and new interactions between trusted hardware and software components. Our techniques are complementary to those that prevent exploitation of bugs in existing software, such as run-time monitoring; they also address different threats and usage scenarios than architectures like TPM that use attestation and measurement of all software.

We focus our efforts on providing enforcement of access control policies, using trusted software to handle the interpretation of complex policies and the protection of data and cryptographic keys. We consider usage scenarios for remote-trust where the local user is trusted with access to certain data, yet the computing device is susceptible to attack. Thus we are able to avoid many of the pitfalls and shortcomings of digital rights management systems.
Chapter 3

Authority-mode SP Architecture

3.1 Introduction

We present a model for trust in portable computing devices, where a central authority manages many devices used remotely. It wants to share secrets and sensitive data with users who are given the devices, but must maintain control over how and when these secrets and data are used. We define transient trust as the ability to trust users with access to protected information for a limited time under certain conditions. We protect information by encryption and hashing, and hence reduce the problem of transient trust to that of secure key management and revocation. Secrets are only available to the users when they have access to the keys that protect the data. In addition to secrets it owns, the authority also wants to provide access to third party secrets on its devices. We define the new authority-mode SP architecture [51] to support such trust in remote devices by adding a few fundamental security features to commodity processors at very low cost.

While this trust model applies to many usage scenarios, as described previously, we use crisis response as a concrete motivating example for this chapter. First responders are provided transient access to sensitive information while in the field. Whether it is a natural disaster, a terrorist attack, a building fire, or a medical emergency, first responders need immediate access to sensitive information stored in electronic databases maintained by a central trusted authority or by other third-party data providers. This might include data about building occupants, medical records, floor-plans, building or city evacuation plans, satellite maps, and other types of information. This sensitive information must be protected at all times, and access should be limited in scope and duration. During an emergency however, the responders are given additional access to sensitive information, which they are expected to use responsibly; under normal circumstances, when having access is less critical, the ability to make specific access control decisions reverts back to the authority and only more restricted access is granted to the responders.

The fundamental hardware features we propose are used to protect critical secrets (e.g., a master cryptographic key and a root hash of a secure storage structure) and to provide a secure execution environment for the software that operates on those...
critical secrets. These hardware features can be used to support software architecture, protocols, and storage that provide important new security functionality. Hence, the processor itself provides hardware-rooted trust for flexible software architecture and usage models. We leverage the secure execution environment from our previous work on the SP (Secret-Protection) architecture [102, 116] to protect the intermediate data generated during the execution of trusted software. But the similarity with the original SP ends there. Our authority-mode architecture supports remote trust usage models with secrets and devices owned by the authority; this cannot be done by the original SP architecture which only provides local trust of users’ secrets on their own devices. We therefore refer to the original work as user-mode SP as compared to authority-mode SP for this new architecture.

A primary contribution of this work is the demonstration of the minimal hardware roots of trust needed for important and complex usage models involving cryptographic access control to protected information. We show that only two hardware registers are needed: a Device Root Key and a Storage Root Hash for the key chain and associated policies. We show how this can be used by a trusted authority: to establish a remote trust relationship with a device it owns that is used in the field (remote trust); to support owner-controlled policy enforcement for the use of secrets on the device (policy-controlled secrets); to provide transient policies that control access to secrets (transient trust); and to securely delegate use of third party secrets to the device user (controlled transitive trust).

This chapter is taken significantly from [51].

### 3.2 Trust Models

Our basic trust model is shown in Figure 3.1, where an authority is an entity that owns many SP devices used in the field. It establishes a trust relationship with each device and can delegate control to the device itself, depicted as arrows in the figure. The authority can distribute secrets (i.e., keys, in this work) to the devices for remote use. It specifies access control by sending policies associated with the keys, which the devices will enforce. Through this model, we explore three concepts for authority-owned devices.

First remote trust, where authority-owned secrets are provided to a remote device with assurance they will be protected using the authority’s own trusted software which it had earlier installed in the device. The trust relationship is based on a shared secret, binding together the authority’s secrets with its trusted software and allowing the device to remotely attest to the authority that it is operating correctly. Second, policy-controlled secrets are the binding of secrets with access control policy for remote enforcement, and their protection from illegitimate modification or separation from each other when stored on the device. We ensure that secrets are only used in authorized ways and only through controlled interfaces. Third is transient trust, which we define as the ability for the authority to vary the level of access given to trusted remote users, based on the severity of the situation, using reliable revocation mechanisms. The trusted remote users are temporarily authorized to make
access control decisions. The authority can later revoke that authorization with more restrictive policies enforced on the remote user. At any time, the authority might change policies, set limits on use, or remove secrets entirely. Such policy enforcement and changes must be guaranteed.

We will also explore a fourth concept, controlled transitive trust, expanding our trust model to include third party secrets and sensitive information. In Figure 3.2, we show a number of third parties who each have a trust relationship established with the authority. The authority and each third party first agree on security policies to be used by the authority’s devices to access third-party confidential data. The authority installs these policies in the devices, which can be trusted to enforce them. The authority thus enables transitive trust from the third parties to individual devices, by establishing policy-controlled temporary relationships between them, depicted as arrows with dashed lines. During operation, the devices can communicate directly with the third parties and access secrets and data for which they will also enforce the associated policies.
3.3 Threat Model

3.3.1 Assumptions

As outlined in Section 1.2.1, our threat model covers operational threats and not developmental threats. Hence, we assume that the devices are manufactured correctly in a trusted factory and that the hardware is free from defects. The authority will receive the devices from the factory with all processor and system features intact and unmodified. Similarly, while we do not assume the regular system software is correct, we assume that the authority’s trusted software is carefully designed and well-tested to ensure that it is correct and has no software security vulnerabilities. Our testing framework, described in Chapter 5, can be used to verify this assumption. Furthermore, we assume strong cryptography for our encryption and hashing, which is computationally infeasible to break.

We assume that the processor chip is the physical security boundary for the hardware. If adversaries have physical possession of a device, they can probe buses on the board but cannot probe inside the processor chip without rendering it unusable. A microprocessor chip is fabricated with many physical layers and processing steps. Because of its complexity, probing of this chip is extremely likely to destroy circuitry unless very expensive equipment is used.\footnote{As described by Anderson and Kuhn [5], very powerful adversaries could extract key material from inside a chip. We generally only consider less well funded adversaries, depicted as Class I or Class II by Anderson and Kuhn [5]. Physical tamper-resistance can be added to the processor chip, to the desired level of protection against such adversaries.} Hence, any registers or cache memory on-chip are assumed to be safe from physical threats of observation or modification. In particular, we add new hardware components to the processor chip, which can only be accessed through the software interfaces that we define. We do not consider side channel attacks on the processor hardware.

We only consider the design of the authority-mode SP client device in this work; we assume the trusted authority has secure systems where it can store its secrets and run its own software with perfect secrecy and access control. Similarly, we assume the authority has a secure depot at which it can initialize new SP devices. Also, while we protect the confidentiality and integrity of the secrets and sensitive data, we do not defend against Denial of Service attacks.

Users are trusted to protect their authentication tokens (e.g., passwords) when logging in to an SP device. They are expected to log out promptly when done with the session. The adversary has no authorized access and will not be able to login to the device.

3.3.2 Threats

The threats for authority-mode remote trust models are quite different than those for a local trust model like in user-mode SP [102], where the user is both the owner of the device and maintains physical possession of the device. The devices and secrets are authority-owned, so while the local users are intentionally given access to secrets, the
users can also potentially be adversaries. Even legitimate users must not be allowed to exceed their authorization and access to secrets, or to use secrets in unauthorized ways.

The main threats we consider are violations of the confidentiality and integrity of keys and policies that the authority supplies to the user of the authority-mode SP device. We also consider threats that affect how the trusted software applies the policies to the keys. Any data released by trusted software to untrusted software is not protected by our design (for this chapter). We assess and protect against both software and physical attacks.

Attackers can launch software attacks on code and data via the operating system, which may have vulnerabilities, or via untrusted applications. They can also launch network attacks by observing or modifying traffic on the public networks. Code and data on disk and in memory, data in transit across the network or buses, as well as the processor state at interrupts, are all vulnerable to spoofing, splicing, and replay attacks. Spoofing introduces false data and modifications; splicing rearranges real data; and replay reintroduces previously used data that had been modified, deleted or revoked. System behaviors controlled by or directed to the operating system are similarly vulnerable to attack, such as virtual memory, I/O devices, processes, scheduling, and interrupt handling.

We also protect against physical threats since an SP device can be lost or stolen. Attackers may have temporary physical access to the device, returning it without evidence of tampering, or “permanent” access on obtaining a lost or stolen device. Thus, these adversaries can mount physical attacks on both the hardware and software. Any code or data stored on the hard disk or in main memory, belonging to the operating system or an application, is fully accessible to the adversaries. By modifying the operating system, they can access and manipulate the full state of an application, including the processor registers during interrupt handling. Main memory can also be accessed by physically probing the memory bus, or by performing rogue DMA operations. I/O buses, including display, disk, and network are similarly vulnerable to physical attack during operation.

3.4 Architecture

In this section, we describe the design of the authority-mode SP architecture. We start with the new hardware components that provide our roots of trust, and then describe how the roots of trust are used to protect trusted software. Finally, we describe how trusted software uses the new hardware features to protect sensitive data and to communicate with the authority, and then how the device is initialized. Later sections analyze the security of the architecture (Section 3.6) and of the cost and performance of the architecture (Section 3.8).
3.4.1 Hardware-Rooted Trust

Our design for authority-mode SP shows that very little hardware support is needed to enhance secure key management and enable transient access. Only two new processor registers containing hardware roots of trust, and a secure BIOS for bootup, are needed to support this trust model. Figure 3.3 shows the architecture of a processor chip with these components added in bold. We use a 128-bit non-volatile register to store the Device Root Key (DRK), and a 256-bit non-volatile register to store the Storage Root Hash (SRH). There is also a 1-bit DRK Lock flag which prevents software from writing to the DRK. We have a secure BIOS to allow initialization of the DRK and to lock it before loading the regular BIOS or other software.

These two registers may only be accessed by a Trusted Software Module (TSM), whose execution is protected by the hardware by a Concealed Execution Mode (CEM). CEM prevents leaking of intermediate values during TSM execution as described below.

In Figure 3.3, the white components represent a typical unmodified processor, and the non-bold shaded components show the SP features added to provide the concealed execution environment for trusted software modules. New instructions to access the SP registers, derive keys from the DRK, and provide the Concealed Execution Mode are listed in Table 3.1. These will be described in detail in the rest of this chapter. A full specification for the authority-mode SP architecture is available in the SP Processor Architecture Reference Manual [52].

One additional register, the CEM Buffer, is added as an intermediate on-chip buffer to hold data moving in and out of the new 256-bit SRH register. Otherwise, writing the SRH would have required two instructions, each moving two 64-bit values from general registers at a time. Instead the data is first placed into the CEM buffer using multiple \textit{GR.Get} instructions. Then a single instruction, \textit{SRH.Set}, sets the entire 256-bit register from the CEM buffer in a non-interruptible, atomic operation. This ensures that there is never an invalid partial value saved in the non-volatile SRH register. The CEM Buffer is also beneficial when generating derived keys from the DRK. This operation generates a 128-bit derived key, while typically an instruction
can only write to a single 64-bit general register. Instead the result is written to the CEM Buffer and read out using \textit{GR\_Set} in word-sized increments.

### 3.4.2 Trusted Software Modules

In our architecture, a \textit{Trusted Software Module (TSM)} is the only software able to directly access the DRK and SRH root secrets stored on the device. This high assurance, trusted software module is provided by the trusted authority who initializes the DRK and SRH in the device at its depot. The authority trusts its TSM to use the secrets correctly and to maintain confidentiality.

The DRK is used to \textit{sign the TSM} by inserting a cryptographic keyed-hash into each cache line of code, upon installation of this trusted code on the device. Later, the code is dynamically verified for integrity during TSM execution (\textit{Code Integrity Checking}), which we describe in detail below.

Both the trusted software and the authority’s secrets will be bound to the DRK, and consequently to each other. Changing the TSM on the device, by anyone other than the authority, requires replacing the DRK, since knowledge of the DRK is necessary to sign new code. This will simultaneously cut off all access to the secrets bound to the previous DRK. Therefore the secrets and the TSM that operates on them are bound together and to the device itself.

### 3.4.3 Secure Execution Environment

To provide protection of the TSM, we leverage the secure execution environment introduced with the user-mode SP architecture \cite{102}. We summarize this as two parts: \textit{Code Integrity Checking (CIC)} and \textit{Concealed Execution Mode (CEM)}. While the hardware support provided for key management in the original user-mode SP architecture does not meet our needs for remote and transient trust, the hardware support it provides for a secure execution environment for trusted software is entirely appropriate, and we describe this briefly. This section describes the shaded, non-bold components in Figure 3.3 and the second group of instructions in Table 3.1.

The combination of dynamic Code Integrity Checking of the TSM with the CEM protection of intermediate data in registers, caches and memory, provides a secure execution environment for executing trusted software that does not leak the values of the DRK and SRH registers, nor the keys they protect.

The first component, Code Integrity Checking, ensures that the TSM code cannot be modified during storage, transmission and execution. It also ensures that no other code can be used as a TSM with access to authority secrets. As shown in Figure 3.4, TSM code is signed by computing a keyed cryptographic hash (i.e., a MAC — Message Authentication Code) over each cache-line of code, keyed with the DRK, and embedding the hash into the code itself. For example, a 64-byte cache

\footnote{The MAC for each cache line is computed over both the code and the virtual address of the line within the application’s memory, to prevent code splicing attacks \cite{102}. The MAC is also computed over a flag to indicate if the line is a valid entry point that can follow a \texttt{Begin\_CEM} instruction (see Section 5.5).}
### Table 3.1: New Instructions for Authority-mode SP

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New Authority Mode SP Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DRK_Set Rs1,Rs2,sel</strong></td>
<td>Set the selected double-word region, ( sel ), of the Device Root Key register from general registers, concatenating Rs1 and Rs2. Requires ( DRK_Lock = 0 ).</td>
</tr>
<tr>
<td><strong>DRK_Lock</strong></td>
<td>Locks the DRK register, setting ( DRK_Lock = 1 ). DRK_Set can no longer be used until the next reboot.</td>
</tr>
<tr>
<td><strong>DRK_Derive Rs1,Rs2</strong></td>
<td>Derives a new key from the DRK by computing a keyed hash over the contents of Rs1 and Rs2. The result is stored in the lower 128-bits of the CEM Buffer register. Available to TSM only.</td>
</tr>
<tr>
<td><strong>SRH_Get</strong></td>
<td>Copies the SRH register into the CEM Buffer register. Available to TSM only.</td>
</tr>
<tr>
<td><strong>SRH_Set</strong></td>
<td>Atomically copies the CEM Buffer register into the SRH register. Available to TSM only.</td>
</tr>
<tr>
<td><strong>GR_Get Rs1,Rs2,sel</strong></td>
<td>Retrieves two words from the general registers into the selected double-word region, ( sel ), of the CEM Buffer register, concatenating Rs1 and Rs2. Available to TSM only.</td>
</tr>
<tr>
<td><strong>GR_Set Rd,sel</strong></td>
<td>Sets a general register with the selected word, ( sel ), of the CEM Buffer register. Available to TSM only.</td>
</tr>
<tr>
<td><strong>Instructions for providing a Concealed Execution Mode for TSM code</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Begin_CEM</strong></td>
<td>Enters CEM for the next instruction.</td>
</tr>
<tr>
<td><strong>End_CEM</strong></td>
<td>Exits CEM for the next instruction. Available to TSM only.</td>
</tr>
<tr>
<td><strong>Secure_Load Rd,Rs,imm</strong></td>
<td>Secure load from memory, at address ( Rs + imm ), to a general register, Rd. Reads from a “secure”-tagged cache line or from main memory with decryption and MAC verification using the DRK. Available to TSM only.</td>
</tr>
<tr>
<td><strong>Secure_Store Rd,Rs,imm</strong></td>
<td>Secure store to memory, at address ( Rs + imm ), from a general register, Rd. Writes to a cache line with the “secure” tag set. Eviction from cache will cause encryption and MAC generation using the DRK. Available to TSM only.</td>
</tr>
</tbody>
</table>
line would contain 48-bytes of TSM code, followed by a 16-byte hash (e.g., using AES-CBC-MAC as the hash algorithm [130]). When the trusted authority installs the TSM, it adds these hashes using its copy of the DRK. Without access to the DRK, no other party can sign TSM code for this SP device.\(^3\)

During execution, the TSM code is verified as it is loaded into the on-chip Level-2 (L2) cache in the microprocessor chip. The keyed hash is recomputed over the cache line of instructions and compared to the stored value, causing any modifications to be detected dynamically during program execution — and not just upon program launch as in other schemes [174, 87]. If the hash check passes, the embedded hashes are replaced with no-op instructions so as not to affect execution. Within the processor chip, the hardware keeps track of TSM code using cache-line tags added to the L2 and L1 caches. Verified instruction cache-lines are tagged as “Secure Instructions” and are read-only. The tag is carried over from L2 cache to the L1 instruction-cache, and is checked when fetching TSM code for execution.

The second component is Concealed Execution Mode (CEM), which protects the confidentiality and integrity of data used by the TSM during execution. Intermediate data is stored in the general registers of the processor, in the caches and in main memory.

Intermediate data that could leak secret information is actively protected by the TSM by using special Secure_Load and Secure_Store instructions to read and write such data. In general, within the security perimeter of the microprocessor chip, secure data is tagged in on-chip data cache lines with a “Secure Data” tag. When such a data cache line has to be evicted from the on-chip L2 cache, it is first encrypted and hashed with the DRK. Similarly, the contents of a data cache line are only hash-verified and decrypted when it has to be brought on-chip into the L2 cache. This happens on only a few Secure_Load instructions which miss in both the L1 and L2 on-chip caches.

Secure data in cache can only be read or written to by TSM code in CEM. Any accesses to “Secure Data” tagged cache lines with normal load and store instructions by either TSM or non-TSM code will cause them to be evicted (encrypted and hashed),

\(^3\)The authority will change the DRK if it upgrades the TSM on an SP device, to ensure that the old TSM code is no longer valid.
and then reloaded in encrypted form. Consequently, a TSM must be written to use each cache line of data for either exclusively secure data or regular data, but cannot mix both types in the same cache line without causing data corruption.

The contents of general registers must also be protected during a processor interrupt. The executing CEM thread can be interrupted at any time for a hardware interrupt or software exception. The OS interrupt handler then saves the process’s register state to memory before executing other code. When an interrupt occurs during CEM, the hardware protects the registers before turning control over to the OS interrupt handler. It encrypts the registers as a single plaintext chunk using the DRK, splits up the resulting ciphertext, and places it back in the registers.\footnote{The use of an Initialization Vector (IV) may also be desirable, depending on the cipher and cipher-mode (e.g., AES-CBC), to prevent leaking of information about identical ciphertext blocks. For the encryption of general registers on an interrupt, the IV should also be saved in an on-chip register or could be passed to the OS to save as if it were an additional register. For secure memory, an IV can be based on the virtual address of the cache line so that repetition of ciphertext does not occur across different data lines. Other memory integrity schemes, including Secure Areas described in Section 5.4.3, can store changing IVs for each secure region over time.} It computes a hash over the resulting ciphertext which is stored in the processor (in the Interrupt Hash register shown in Figure 3.3), along with the memory address of the next CEM instruction (saved in the Int. Address register) and context information (i.e., process ID and context ID, if provided to the hardware by the OS). These are checked automatically to resume CEM when the interrupt is completed. With the contents of the general registers thus protected, the hardware can then allow the OS interrupt handler to assume control to save and restore the register state without being able to read the (plaintext) contents. The processor then watches all instruction that could return from the interrupt (e.g., explicit RFI instructions or jump instructions). It compares the destination address and context information to the saved values, and resumes CEM if there is a complete match; it must also verify and then decrypt the general registers, raising an exception if the verification fails.

The state of the processor, whether or not executing in CEM and whether a thread in CEM has been suspended, is indicated by the CEM Mode bits in Figure 3.3. Lee et al.\footnote{Lee et al. [102] provide more details on CEM and the hardware support for a secure execution environment.} provide more details on CEM and the hardware support for a secure execution environment.

### 3.4.4 Secure Storage for Keys

Building off of the secure execution environment provided by CEM, the TSM in turn is responsible for protecting persistent secrets (e.g., the authority’s keychain) from unauthorized access. For this protection, we can create secure local storage, a data structure managed by the TSM that provides hierarchically encrypted and hashed storage of keys.

This secure storage structure incorporates a Merkle hash-tree mechanism\footnote{Lee et al. [102] provide more details on CEM and the hardware support for a secure execution environment.}, storing the root hash in the Storage Root Hash (SRH) register on-chip. The root hash, updated only by the TSM, ensures integrity of the keys against malicious modifications by untrusted software. Since the root hash is stored securely on-chip, the...
secure storage is also protected against replay attacks — changes to the secure storage structure are made permanent, including deletions. Stale or deleted data cannot be replayed since the root hash will no longer match.

Replay-resistant secure storage can provide transient trust for the authority’s key-chain. At the authority’s request, the TSM can permanently revoke access to certain keys by deleting them from the secure storage. The secure storage also contains nodes for access control policies associated with keys, which the authority can update at any time and are enforced by the TSM. This enforcement provides policy-controlled secrets, which the authority can use to cut off or limit access when a predetermined condition is met. Thus it provides another means of revocation that is effective even when the authority cannot communicate with the remote device.

The basic tree structure of the secure local storage is shown in Figure 3.5(a). All non-leaf nodes are directory nodes (DNs) and store special meta-data about their child nodes. Leaf nodes can store not only keys, but also data, additional meta-data, and access control policies. Typical items are shown in Figure 3.5(b); each is actually a collection of nodes: a directory node and leaf nodes which then contain keys or data along with their respective policies and meta-data.
The secure storage structure is built on top of insecure OS storage facilities and must be protected from attacks while on disk or in main memory. Each node in the structure is encrypted with a derived key, generated from the DRK using a nonce stored in its parent DN. A different derived key, using the same nonce, is used to generate a keyed-hash (MAC) of the contents of the node. The resulting Merkle hash-tree is incorporated into the secure storage structure by storing the MAC in the DN. When data is first added, a random nonce is chosen; it is then saved to regenerate the same derived key for later decryption and integrity verification. If a unique nonce is chosen for each DN entry, then each node in the tree will be encrypted with a different key.

The DN structure, in Figure 3.6, also contains meta-data identifying the nodes, their type, size and location on disk. A chain of IDs can trace a path through the tree to reach a particular node, such as “0:2:7” for node 7 in Figure 3.5(a). When the TSM reads a DN, it can use the type field to quickly identify any meta-data and policy nodes that apply to a key or data node without decrypting each node. Once found, policies are enforced by the TSM and are inherited hierarchically from parent to child; conflicting policies are enforced by allowing policy nodes lower in the tree to override general policy set by a parent or ancestral node. Any descriptive or application-specific meta-data (e.g., cipher type, creation date) can be stored in a separate meta-data node.

Data nodes and key nodes are distinguished with different types (shown in Figure 3.6(a)) so that specific policy can be set to control keys, preventing them from being leaked as data. The policy node can contain an Access Control List (ACL), indicating which users are allowed access to the data or key, and under what conditions. The policy may include generic read/write permissions, limits on the number of accesses, expiration dates, or type-specific access control, such as which queries
can be made to a database. Policy nodes can also contain general configuration for the authority’s software, such as user customizable settings. The format for policy files is determined by the authority’s software and can be implementation specific. Secrets used for user authentication are similarly stored in the tree.

Figure 3.7 shows a simple storage tree with only one level of items. At the root of the tree is a special DN, the Root Directory Node (RDN). Unlike all other DNs, the RDN does not have a parent to store its directory entry. Instead, the TSM is hard-coded with a location to find its nonce and size (both unencrypted), and the encrypted contents of the node. The root MAC is stored in the on-chip SRH register, where it is protected from attacks.

3.4.5 Derived Keys

Secrecy of the secure storage is rooted in the DRK. Since the DRK is a critical secret used to maintain the trust relationship with the authority, it must never be revealed to any software on the device. Even the TSM is not permitted to read the DRK. Instead the TSM derives new keys from the DRK using a new processor instruction, \textit{DRK.derive} (see Table 3.1). This instruction performs a keyed cryptographic hash function to combine the DRK with a value or nonce provided by the TSM.\footnote{Since the nonces are provided by the TSM, the hardware does not need to include a random number generator. Similarly, keys are provided by the authority or generated as derived keys by the TSM, and do not require hardware support for generation.} These derived keys are used to encrypt the nodes of the secure storage structure. As a result, the information in secure storage is bound to the device and cannot be read (decrypted) or modified without detection by non-TSM software.
For more general purpose use of derived keys, the value provided to the hardware can include a constant that will distinguish keys used for different purposes, e.g., communications versus storage. It also includes one or more nonces to make each key unique. When the software needs to regenerate a particular key, such as an encryption key for secure storage, the nonce can be saved with the data and reused by the TSM to obtain the same derived key. Similarly, if the authority and the device need to produce the same key, they can exchange nonces and generate the same derived key — the authority deriving the key from its copy of the DRK in a secure environment. On the device, the TSM is responsible for protecting the secrecy of the derived keys that it produces, however the constants and nonces need not be secret.

As an example, Table 3.2 shows that derived keys for storage will be derived using a constant, $C_{\text{Storage}}$, and a different nonce for each storage node. An additional set of constants, $C_{\text{Enc}}$ and $C_{\text{MAC}}$, distinguish keys used for encryption and hashing of storage nodes. Derived keys that will be used as session keys for secure communications and remote attestation can be derived using a different initial constant, $C_{\text{Comm}}$, along with nonces that ensure freshness from the authority, $N_A$, and the device, $N_D$. Additional constants, $C_A$ and $C_D$, provide different keys for each direction of communication.

To generate a derived key using the SP hardware, the TSM combines the constants and nonces in software into a single 128-bit nonce by hashing them. Then the $\text{DRK}_{\text{Derive}}$ instruction is used to create the derived key in hardware with the DRK and this nonce. This is repeatable and will always produce the same derived key for a given nonce, as long as the DRK remains unchanged.

Table 3.2: Derived Keys for Storage and Communication

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Derived Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage key for encryption of node $i$</td>
<td>$K_{S_i\text{enc}} = MAC_{\text{DRK}}[C_{\text{Storage}}, C_{\text{Enc}}, N_{S_i}]$</td>
</tr>
<tr>
<td>Storage key for hash of node $i$</td>
<td>$K_{S_i\text{MAC}} = MAC_{\text{DRK}}[C_{\text{Storage}}, C_{\text{MAC}}, N_{S_i}]$</td>
</tr>
<tr>
<td>Comm. key for Authority to Device</td>
<td>$K_{A\rightarrow D} = MAC_{\text{DRK}}[C_{\text{Comm}}, C_A, N_A, N_D]$</td>
</tr>
<tr>
<td>Comm. key for Device to Authority</td>
<td>$K_{D\rightarrow A} = MAC_{\text{DRK}}[C_{\text{Comm}}, C_D, N_D, N_A]$</td>
</tr>
</tbody>
</table>

### 3.4.6 Remote Attestation and Secure Communications

For an authority-owned device, we define remote attestation as the process of proving to the authority that the device still has possession of the correct DRK and is still running the correct TSM software that properly uses and protects the authority’s keys and policies. Remote attestation is necessary when establishing communication channels between the authority and its device in the field. Note that this is a different and more lightweight “attestation” than that used in TPM-based systems [174], where remote attestation means verifying the integrity (cumulative hashes) of every level of software running on the machine, up through the desired application level.

To establish communications, the parties first need a secure channel over the public untrusted network. This is done by generating a session key which can encrypt and
Communication Local Computation

A (Authority), D (Device)

Initiate communications, generate nonces & send challenges:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → A:</td>
<td>ID_D; D: send device ID to initiate communications</td>
</tr>
<tr>
<td>A → D:</td>
<td>N_A; A: generate random 128 bit nonce</td>
</tr>
<tr>
<td>D → A:</td>
<td>N_D; D: generate random 128 bit nonce</td>
</tr>
</tbody>
</table>

Generate derived keys for communication:

\[ K_{A→D} = MAC_{DRK}[C_{Comm}, C_A, N_A, N_D] \]
\[ K_{D→A} = MAC_{DRK}[C_{Comm}, C_D, N_D, N_A] \]

Send responses:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A → D:</td>
<td>MAC_{K_{A→D}}[N_A, N_D]; D: save as Response_A</td>
</tr>
<tr>
<td>D → A:</td>
<td>MAC_{K_{D→A}}[N_D, N_A]; A: save as Response_D</td>
</tr>
</tbody>
</table>

Verify responses:

\[ A: MAC_{K_{D→A}}[N_D, N_A] = ? Response_D \]
\[ D: MAC_{K_{A→D}}[N_A, N_D] = ? Response_A \]

Figure 3.8: Challenge-response Protocol for Generating Session Keys and Attestation

hash all messages. The session key will be a derived key based on the DRK, which is a secret shared only by the authority and this particular SP device. No other parties, except the authority and the device’s TSM, can produce the correct session key. Consequently, using a correct derived key proves knowledge of the DRK, so setting up a working secure channel implicitly serves as remote attestation for our purposes.

Since no other party knows the shared secret, attestation also serves as a form of mutual authentication, in the sense that the device and authority both know that the other has the same DRK. The authority knows it is communicating with a good TSM on a device that it can trust. It can send new secrets and policy, knowing they will be properly protected by its own signed TSM. It is also assured that any data or messages sent back from the device, through this secure communication channel, must have been generated by its TSM. Similarly, the device has authenticated the authority, so it can incorporate new keys or policies sent over this secure channel.

Secure communications can be established between the device and the authority using the challenge-response protocol given in Figure 3.8. Either side can initiate the protocol, during which both sides select a nonce and transmit it in plaintext. The nonces are combined with the constant prefixes to generate the derived session keys, \( K_{A→D} \) and \( K_{D→A} \). Once these keys are established, the parties each send a MAC of the combined nonces, using their session key. Verifying these MACs confirms that the authority and device share the same keys and that no man-in-the-middle or replay attack has occurred, since no other party could have known the DRK in order to generate a valid MAC. This verification requires each side to generate both of the uni-directional keys (in the middle section of Figure 3.8), using one key to send messages, and another to decrypt and check received messages. These session keys
can be used directly for secure communications. Alternatively, they can be used in a standard protocol such as TLS-PSK [57], which initiates the TLS protocol using pre-shared keys.

3.4.7 Device Initialization

The authority initially establishes trust in a device at its trusted depot, using physical access to install the DRK. This new secret is generated randomly by the authority for each device and is therefore independent of the manufacturer, any other device, and any past use of the device. The authority saves a copy of the DRK in its own database along with the device’s ID. It then boots the device into the secure BIOS by setting a hardware jumper. The secure BIOS bypasses normal bootup and executes a verified initialization routine from its ROM, which executes entirely within the security perimeter of the processor chip, using only on-chip caches. The routine saves the new secret in the on-chip DRK register. The device is rebooted, removing the jumper, and from then on skips the initialization routine. Instead the secure BIOS executes only the DRK_Lock instruction before passing control to the regular system BIOS for a normal bootup.

Then the TSM must be signed with the new DRK. This can be done on the authority’s secure computers, where the verified TSM code resides. The signing process computes hashes over each cache line using the DRK, and stores them in empty spaces left by the compiler. The signed TSM, and other system code, can be copied to the disk or flash storage on the device.

Finally, the authority initializes the secure storage structures; for our usage scenario, this stores the keys and their associated usage policies and meta-data. The authority must also set up user authentication and authorization data, storing user passwords, biometric data, or users’ cryptographic keys and specifying what privileges each user has. The authority builds the secure storage tree using derived keys, and then uses the TSM to write the initial value of the SRH register. All of the policies and secrets can be updated in the field via secure communications.

3.5 Usage Scenarios

Our new remote trust model is applicable to many different scenarios, where an authority wants to extend trust to remote devices it owns that are used in the field. Below, we provide a detailed usage scenario for crisis response.

We provide transient access to keys that enable access to protected (encrypted) data when the crisis starts, and revocation of access to those keys when the crisis ends. At all times, we provide confidentiality of these keys. Even if a device is lost...

---

6The secure BIOS initialization routine will not depend on any other software on the device, but will need access to some input device to obtain the DRK value generated by the authority. There is no threat to exposing this value, since the operation takes place in the secure depot and the initialization routine will not save a copy of the DRK elsewhere on the device.
or stolen, the confidentiality of the keys, and hence the sensitive data they protect, must be maintained.

First responders can use their device (e.g., an SP-enabled secure smart phone or laptop) for both critical and non-critical applications. Non-critical applications include use of e-mail, web browsing, instant messaging, voice chat, etc. for uses not related to a crisis. These do not use the TSM and may involve off-the-shelf or downloaded software that must not put at risk the sensitive data made available to crisis responders. For critical applications, the integrity of communications will be critical or sensitive data will be needed, such as command center personnel coordinating a response strategy, firefighters accessing building plans and occupant information, or paramedics accessing medical records when they get to the scene of an accident. For these situations, the authority will distribute secrets and access control policies in advance of a major crisis.

After authenticating, the sensitive information on the device will only be available to the responder through the controlled interfaces of the TSM, which may limit how data is accessed. For example, this might mean that even with a complete medical record downloaded and accessible to the TSM, the paramedic can read a patient’s allergies and medical history but cannot access the psychiatric portion of the record. Rate limiting might also be employed, so a paramedic can access a few, but not hundreds of records at any one time, and those accesses can be audited and reported back to the authority on a regular basis. These policies are all programmed into the TSM.

3.5.1 Preparation

The authority makes general access control decisions in advance for many likely crisis scenarios so the response can be quick when a crisis does occur. For example, to give the firefighter access to floor plans and building occupants, data for all buildings in a city should be prepared in advance. When a crisis occurs, the city’s crisis response authority can decide which data to make available based on which parts of the city are affected. The city authority will negotiate access rights with building owners and hospitals in advance and decide how to delegate rights to individual responders using devices it will distribute. It forms trust relationships with each of these third parties in advance, setting up a certificate authority (e.g., using X.509 certificates [176]) so it can specify access rights and allow devices and third parties to authenticate each other during the crisis.

When a crisis begins, the authority will create and sign certificates for each device, tagged with a crisis ID and a reasonable expiration. The certificate specifies what data should be made available to the device, and the IDs of the encryption keys from the secure local storage that should be used to store that data when it is transmitted to the device; this will determine which policies apply to the data. The authority

---

7The authority’s TSM should periodically re-authenticate a user to reduce the risk from a device that is lost or stolen while a legitimate user is logged-in. The frequency and method of re-authentication should balance the risk of exposing sensitive data against its inconvenience during an emergency.
communicates with each device, which performs remote attestation that it still has the correct DRK and DRK-signed TSM; it then enables pre-distributed keys already loaded on the device and distributes additional keys over the secure communication channel, along with the certificates for third party data and their associated access policies.

### 3.5.2 Crisis Operation

During the crisis, responders can contact third party data sources directly, via their devices, to retrieve sensitive data which they are pre-authorized for by the authority’s certificates. Since the certificates will be stored securely on the device by the authority and signed by it, its possession by the device is sufficient to authenticate the device to a third party. However for more complicated scenarios, the authority can give each device its own public/private key pair, which it can use to authenticate directly to the third party, along with a signed authorization token from the authority specifying what data it can retrieve.

Responders can then make specific access decisions in the field to make use of the sensitive data they retrieved, within the constraints set by the authority, depending on which specific data is needed. Devices can also contact the authority during a crisis to obtain additional keys and access rights while in the field. Unanticipated circumstances will surely occur, requiring access to data beyond what was pre-authorized. For example, “need-to-know” may be determined in the field, for responders to have access to additional data. Alternatively, some data can be sent in advance with specific limits for offline enforcement, to be revealed by the TSM only if needed. Such data would be requested by the responder, with the TSM enforcing a limit to reveal only up to a fixed fraction of that data while offline.

### 3.5.3 Revocation post-crisis

Access to sensitive information must be revoked after a crisis has ended. This is accomplished through a combination of three revocation mechanisms. First, all secrets given to the device are policy-controlled and will include limits on use, including a maximum number of accesses and/or an expiration date. So as not to cut off access while the crisis is still going on, these restrictions must necessarily give a wide safety margin. This first line of defense is effective even when the device is operating off-line.

Second, the authority can directly cut off access by contacting each device to delete secrets and modify the stored policies. The TSM will confirm that the revocation was successful and that the secure storage’s hash-tree and SRH register were updated. The TSM can also report back what accesses had been made during the crisis. Knowing that accesses will be audited will provide a disincentive for abuse by authorized users.

Third, access to new secrets and data will be cut off, both from the authority and third parties. The authority will contact each third party and report that a particular crisis ID is no longer valid, or will revoke certain certificates. Any certificates that specified this ID will no longer be accepted for access, while general certificates for
day-to-day use are still available. A new crisis ID can be used for the next crisis or another phase of the same crisis.

3.6 Security Analysis

3.6.1 Protection of Authority-mode Registers

The security provided by our architecture is rooted in trust in the new processor features. Since the processor chip is in our physical security perimeter, our new authority-mode registers (i.e., the DRK and SRH) can only be accessed through software instructions we define. We do not define any instructions to read the DRK register, and only allow writing by someone with physical possession of the device using the secure BIOS; therefore the DRK register contents can never be observed outside the processor or copied to another device and can never be modified except under secure BIOS execution. Similarly, only TSM software can use instructions for accessing the SRH register and for deriving keys from the DRK. Since the TSM is the authority’s own correct, trusted software, no other software can ever observe or modify the SRH or generate derived session keys from the DRK.

For our usage as an integrity mechanism, the SRH need not be kept confidential. Adversaries with physical possession of the device could overwrite the DRK, sign and install their own TSM, and use that TSM to read the contents of the SRH register that was valid with the authority’s TSM. However, this SRH value is not useful since the new TSM cannot decrypt the persistent storage corresponding to the SRH once the DRK has been changed. If absolute confidentiality of the SRH is needed for other applications, then the hardware should clear the SRH register as part of the DRK_Set

3.6.2 Protection of TSM

The dynamic Code Integrity Checking for the TSM instructions ensures that the code remains unmodified throughout its execution. This defends against dynamic hostile code insertion attacks, in addition to static changes to code on disk. TSM code is verified against the embedded MAC as it is loaded into the on-chip caches, ensuring that any changes are detected. This MAC includes the virtual address of each block of code, such that code cannot be spliced or replayed in different locations. Similarly, the OS is unable to use interrupts and virtual memory mappings to manipulate how TSM code is loaded, as the same virtual addresses used for execution are checked for return addresses and secure memory accesses. Old TSM code cannot be spliced or replayed after a software upgrade from the authority since the authority will change the DRK (and re-encrypt and MAC the secure storage) any time the TSM code is changed legitimately.

Furthermore, Concealed Execution Mode ensures that secure intermediate data and general registers, during TSM execution, cannot be observed or modified without detection. If a modification is detected, the processor raises an exception; the TSM
operation will be aborted and processor registers cleared. Observation of secure data in physical memory is not possible by either the OS or a physical attacker [75], as secure memory is always encrypted when sent off-chip. Therefore the TSM should be written to ensure that all keys and secrets are stored using on-chip registers or secure memory. Together, Code Integrity Checking and Concealed Execution Mode guarantee that a TSM will be fully protected during its execution, and these mechanisms remain roughly unchanged from user-mode SP [102].

However, our authority-mode architecture provides additional trust guarantees over user-mode. Since TSM code can only be hashed with the DRK, which is only known outside of hardware by the authority, no new TSM code can be added to a device by an adversary without changing the DRK. Thus, the authority can check that its unmodified TSM is running by having the device attest to having the same DRK value shared with the authority. If an adversary tries to modify TSM code or install new TSM code by replacing the DRK, the derived keys generated by the new TSM, based on this new DRK, would be different and the device will be unable to authenticate itself and perform remote attestation. Also any data encrypted by the authority’s TSM will no longer be accessible. As a result, protected data written by any TSM is bound to that TSM. No other software is trusted with the authority’s data or with policy enforcement, so integrity checking is only necessary for the TSM.

3.6.3 Protection of Persistent Secure Data

All data from the authority that starts out under control of the TSM will remain secure as long as the TSM handles it properly on the device. The authority has written and tested the TSM to ensure this property, making use of the persistent secure storage structure. In secure storage, confidentiality is protected with DRK-derived keys so that stored data can only be read by the TSM. Integrity is based on a hash tree rooted in the on-chip SRH register. That register can only be modified by the TSM, which is continuously installed as long as the DRK is unchanged. If the TSM were replaced, a new TSM could access the old value of the SRH, however, this value is meaningless without access to decrypt the data in the rest of the local secure storage structure. While the new TSM could change the SRH, the DRK can never be changed back to the original value since it is unknown, except to the authority. This new adversarial TSM will be useless for accessing sensitive data, since the device can no longer attest the old DRK to the authority, nor can it access the secrets stored under the old DRK.

A legitimate TSM can use the SRH to verify the contents of the secure local storage structure. It recomputes a hash over the root directory node (RDN), the root nonce, and the size of the RDN. If this matched the value stored on-chip, it will use the nonce to generate a derived key for decryption to access the RDN, which in turn contains hashes and nonces hierarchically for the rest of the structure. The root nonce and size are stored in plaintext, but provide no useful information to an attacker who cannot change the SRH value on-chip.
3.6.4 Remote Attestation and Communications

Remote trust also involves secure communication and remote attestation, which is based on the authority and device sharing the DRK secret. Remote device attestation (and mutual authentication) requires each side to prove knowledge of the DRK through the challenge-response protocol. Hence, our authority-mode protocol enables the authority to know that the device still has the shared DRK, and enables the device to know that it is speaking to the correct authority, i.e., one that knows the shared DRK. Random nonces prevent replay attacks on the exchange and are also used to generate session keys for communication. Man-in-the-middle attacks are not possible since no other party can generate the correct derived keys, to be used as valid session keys, as they do not possess the DRK.

Once the authority knows that its original DRK is still in the device and was used to correctly generate derived session keys, used for setting up the secure communications channel, it can be certain that only the TSM code that it originally signed with this DRK is running on the device and taking part in the communications. In fact, stronger than the integrity-checking done only on program launch by TPM-based systems [174], both user-mode and authority-mode SP devices provide dynamic Code Integrity Checking during the fetching of cache lines for the TSM throughout its execution. It is safe for the authority to transmit new data to the TSM, knowing it is continuously integrity-checked and protected by the DRK.

3.6.5 Policy-controlled Secrets

In our architecture, keys and policies cannot be separated without modifying or deleting nodes from the secure storage tree, which requires modifying other nodes, and ultimately modification of the root hash stored in the SRH. More generally, spoofing attacks on secure storage would include deliberate modification or insertion of data into the structure. The integrity checking will detect any such changes, and the MAC entries themselves cannot be spoofed without knowledge of a DRK-derived key.

If nodes could be rearranged, duplicated or removed, it would also affect the application of access control policies. Since each node or subtree is self-contained and independently hashed, parts of it can be reused at lower levels. However, the parent of each node is a DN that lists its children and their MACs in order. Any such splicing of the secure storage tree structure would be detected when the MAC of the parent is checked. Since the TSM performs checks all the way to the root SRH, which is on-chip and safe from attack, it is guaranteed to detect any illegitimate changes. Therefore, the authority knows that an access policy stored on the device cannot be modified and will be enforced by the TSM.

3.6.6 Transient Trust

Replay attacks on memory and disk storage are a threat to revocation, which is essential for transient trust. This is especially critical for data in the persistent secure storage tree. If the entire storage tree can be replayed, then any policy updates or
revocation of keys or data performed by the authority could be undone. However, since the root hash is stored on-chip, out of reach of any adversaries, this is not possible; previous trees will not be valid. The only way to add or remove data from the tree is through the TSM, but the TSM will only make legitimate modifications specified by the authority during secure communications or through previously set policy. Once the TSM updates the SRH register with the new root hash, these modifications become permanent.

For offline revocation using pre-defined access control policies, it is also important to have an unspoofable time source to correctly apply expiration dates. We consider this out of the scope of our work, but fall back on the other revocation methods if such a time source is not available. In particular, we are able to guarantee that a device that is compromised after a crisis ends, and has had its policies updated, will not be susceptible to replay attacks that falsify the time in an attempt to gain access to sensitive data that was only accessible during the crisis.

### 3.6.7 Transitive Trust

Transitive trust is based on the TSM’s use of secrets and the authority’s trust relationships established with the device and the third parties. These components are all individually assured, so the TSM’s implementation of transitive trust will be maintained.

### 3.6.8 Other Issues

In addition to the TSM, we expect that other trustworthy system software (e.g., a security kernel and secure I/O drivers) will be present to assist with user authentication and secure display of data to the user. These are orthogonal issues to our design, not considered in our threat model and thus not discussed in this chapter. Without trusting these mechanisms, the authority’s TSM cannot guarantee that data displayed to the user will not be copied by some software. Therefore we place some responsibility on the user — a reasonable tradeoff since users can always reveal the data once it is outside of the device. Users should not intentionally login and access data when they know the system to be physically compromised or suspects that the system software was replaced or compromised. The TSM and policies should set up controlled interfaces to release data to the user in as safe a way as possible, mitigating any risk from data leaked while being displayed. Only data being displayed is vulnerable in this way. Other data controlled by the TSM, that the user could have accessed but did not, is not vulnerable. Other secrets accessed by the TSM (but not displayed) or stored securely on the device will not be leaked.

Our threat model currently does not defend against denial of service attacks, which prevents a guarantee of availability. Attacks on the TSM code, TSM execution, communications protocol, secure storage structure, etc, will all be detectable, but currently we have not defined recovery mechanisms.\(^8\) This is acceptable under our

\(^8\)Where possible, we put in place defenses against certain denial of service attacks. For example, the locking of the DRK register by the secure BIOS prevents software attacks that overwrite the
threat model, which guarantees confidentiality of keys and the data they protect, but is less acceptable for a device relied upon in a crisis. We consequently leave availability for future work, which might require changes to the trust model and the architecture. While our architecture does not require trust in system software to provide remote attestation or protect the authority’s secrets, the best defense for availability is to install a secure OS that can protect itself from untrusted applications and remote exploits, and for the user to maintain physical control of the device.

Our architecture uses strong ciphers and provides hardware support for 128-bit encryption keys and 128-bit keyed-hashes, using AES hardware on-chip. Over time, it will be necessary to increase the key length and/or change algorithms to maintain sufficient security. Our architecture is generally flexible to these modifications with minimal changes, mostly affecting instruction cycle time, cache-miss penalty, and area for on-chip registers. In a few cases, somewhat more significant implementation changes may be necessary, such as signing TSM code in larger blocks if MACs take up too many bits of each cache line; this will again affect the cache-miss penalty.

Similarly, distinct keys should be used in hardware for encryption and hashing, requiring more total key bits for the Device Root Key to derive these keys. In software, the TSM uses different constants depending on the protocol used, always making derived keys specific to their purpose, and thus helping to prevent attacks from key re-use in different protocols.

3.7 Comparison with User-mode SP Architecture

The previous SP [102] architecture, which we call user-mode SP, was designed to protect users’ own secrets and provide portability of their secrets across multiple devices. Portability requires decoupling trust in the device from trust in the users’ secrets (e.g., their encrypted keychains). Users can trust multiple devices independently of where their secrets are stored, and could even access their secrets on unprotected devices. It is the users’ own responsibility to choose where it is safe to access their secrets.

We found the CIC and CEM architectural features of the user-mode SP architecture to be effective for providing a secure execution environment for trusted code and have used them in our new authority-mode SP architecture. However, the key-chain and master secrets (User Master Key, UMK, and Device Master Key, DMK) defined DRK and effectively disable all trusted use of the device; this becomes possible only with physical access. As long as the DRK is not changed, the remainder of the software and secure storage can be restored from a backup to make the device fully functional again after an attack. Also, to prevent corruption of the SRH value, updates to the SRH register are first written to the CEM Buffer register and then copied atomically to the SRH register, guaranteeing that the operation cannot be interrupted when only part of the SRH has been written. This ensures that at all times, the value of the SRH register is valid for either the old or new copy of the persistent secure local storage. There is never a time when partial data was written to the SRH register and no secure data is valid.

9Any such change must ensure that it is not possible to modify the derived key or part of DRK used for one purpose without also invalidating the key(s) used for all other purposes. Otherwise the DRK binding properties of authority-mode SP could be violated. The simplest solution is to use a single (possibly larger) DRK value to derive other needed keys, in hardware.
and protected by user-mode SP, while suitable for a device owned by the user, could not provide the functionality we need for a device owned by a remote authority rather than the local user. This functionality included mechanisms providing remote trust, secrets bound to policies, transient trust guarantees, and controlled transitive trust that our new authority-mode SP architecture provides. Before describing each of these key differences in turn, we first point out an attack on SP.

In user-mode SP, users must maintain physical control of the device to be sure their trusted software module (TSM) has not been replaced by an adversary who gets temporary access and replaces the Device Master Key (DMK). An attacker can change the DMK to DMK’ and install a malicious TSM’ using DMK’. In user-mode SP, users will not be able to detect this — they may unknowingly enter their keys for use on a device where an adversary had installed a malicious TSM’. In the authority-mode SP architecture, the authority can always detect whether its good TSM is still running, hashed by the Device Root Key (DRK), which is the secret the authority shares with the device. By enabling a trusted authority to check the DRK on the device, especially before new keys are sent to the device, authority-mode SP thwarts the attack of some adversary changing the TSM. In Section 7.3, we describe an extension to user-mode SP that addresses this attack, protecting the user’s UMK from being exposed to an untrusted TSM.

Remote trust requires some attestation ability. User-mode SP has no device attestation capability because no mechanism exists to verify the contents of the DMK. In contrast, our authority-mode architecture makes remote attestation possible due to the shared secret (DRK) between the authority and the device. Our new DRK\_derive instruction allows generation of derived keys for new communication sessions, based on this shared DRK secret. The generation of a correct new session key, according to the communications protocol implemented by the TSM using the DRK, proves to the authority that the device still has the correct DRK.

Without binding secrets to a TSM, policy-controlled access is also not guaranteed. In user-mode SP, the user’s secrets can be accessed on any device with any TSM the user chooses to trust. Therefore, there is no guarantee that any policies will be enforced. In our authority-mode SP architecture, we bind secrets to policies in our persistent secure storage structure, and the authority can ensure that its good TSM is still running, hence ensuring that the access control policy specified will be implemented and cannot be violated by the local user.

User-mode SP does not provide transient trust reliably, since this requires permanent revocation of keys. Since user-mode SP does not bind the users’ secrets (e.g., their keys) to one device, intentionally allowing the user’s key-chain to be usable on multiple devices, one device will not know of changes or accesses made on a different device. Furthermore, while user-mode SP takes care of register replay attacks during interrupts and exceptions of the TSM, it does not provide protection from storage replay attacks that prevent from using a key that has been revoked (i.e., deleted from the user’s key-chain structure).

Without a hardware root hash (as in authority-mode’s SRH register), no changes can be made permanent, even on a single device — the entire key-chain in user-mode SP can be copied at first and later replayed after changes are made. Because any
hashes for integrity protection can be replayed along with the data, a user-mode SP device cannot tell that the data is stale and had been modified. Integrity checking will still ensure arbitrary modifications are not made, but any keys that were once valid will always be valid, making revocation impossible unless the user’s root secret (the User Master Key in user-mode SP) is changed. Our authority-mode SP architecture provides reliable revocation of keys in the persistent secure storage structure, hence enabling transient trust.

Lastly, user-mode SP does not provide transitive trust because there is no remote trust mechanism at all. While a third party can give secrets to the user directly, it has no guarantee from another trusted party (e.g., an authority) that its secrets will be protected or used in a particular way. The third party can also supply its own TSM with its secrets directly to a user-mode SP device, but again has no guarantee that this TSM will not be modified or replaced by either an adversary or the device user himself. Furthermore, a user-mode SP device only protects confidentiality of secrets with the user’s root secret, so the third party’s secrets cannot be hidden from the user.

In summary, while our authority-mode SP architecture leverages the minimalist design philosophy and some features of user-mode SP, it provides significant new security features and defends against some potential attacks on user-mode SP. However, it requires device users to trust an authority to set up their device in a secure depot, and does not support portability of the users’ keychains across multiple devices or privacy of the users’ keys from the authority.

### 3.8 Cost and Performance Analysis

#### 3.8.1 Cost

The authority-mode architecture is implemented in hardware with only five new registers (totaling 880 bits) plus 2 state bits and 1 lock bit (see Figure 3.3). The five registers are the new 256-bit SRH register, 128-bit DRK register, and 256-bit CEM Buffer register of authority-mode, and the 64-bit Interrupt Address (plus it’s corresponding 32-bit PID and 16-bit VMID fields) and 128-bit Interrupt Hash registers to provide the Concealed Execution Mode common to both user-mode SP and authority-mode SP. For on-chip caches, an insignificant 1-bit cache tag per L1 cache line and 2-bit cache tag per L2 cache line is added. An encryption/hashing engine is added at the chip boundary. Eleven new instructions are defined, and a secure BIOS is required. This represents a tiny and insignificant cost for any commodity microprocessor or SOC (System-on-Chip), and even for many embedded processors.

#### 3.8.2 Performance

Non-TSM software is basically unaffected by the new hardware since it will never trigger CEM protection or access the new registers or instructions. Furthermore, the security of authority-mode TSMs and secrets does not depend on limiting other
software or verifying the entire software stack. Therefore, there is a negligible performance impact for system software and applications that do not use the TSM.

TSM software will incur a slight performance penalty (see \cite{116, 102}) — McGregor et al. \cite{116} found this penalty for a very similar architecture to be less than a 1% delay for a benchmark TSM that implements cryptographic functions. Our designs will lead to somewhat larger TSMs, but we expect the effect on performance to be comparable. Furthermore, the degradation of performance only impacts the small TSM module where security is critical and a slight delay is acceptable.

One source of the delay is for Code Integrity Checking (CIC) of TSM code and Concealed Execution Mode (CEM) checking for secure loads and stores. These both occur only at the cache-memory boundary upon the rare L2 cache\textsuperscript{10} miss, where the miss penalty is already several hundred cycles for typical microprocessors. Hence, some tens of extra cycles for hardware hash computation and symmetric-key decryption or encryption is not going to cause much performance degradation. Once inside the L2 and L1 caches, accessing secure instructions and data proceeds as fast as before, since the checking of the 1- or 2-bit secure tags for a cache line (that has already been verified) is very fast and may even add zero delay if done in parallel with other necessary checks.

CIC checking inserts additional no-op instructions into the instruction stream for each cache line. These will cause some degradation in the efficiency of the instruction-fetch process, since a fraction of instructions fetched are useless, thus somewhat reducing the instruction fetch bandwidth and cache locality. In a modern out-of-order processor, this should have insignificant effect on execution throughput. CEM interrupt protection requires the encryption and hashing of the general registers, but interrupts are relatively infrequent compared to the other factors.

The final component affecting performance is the design of the TSM software itself. Rather than directly accessing unprotected secrets, the TSM will perform additional cryptographic operations to retrieve secrets. For example, navigating the storage tree requires traversing nodes in the tree, each of which requires generating two derived keys in hardware which are used to check hashes and decrypt the node with software operations. Additional data may have to be retrieved from disk, involving the OS to access the file system, therefore causing additional system calls and memory/disk accesses, and possibly affecting cache behavior. We have designed the secure storage tree navigation to predominantly use symmetric-key operations which are significantly faster than asymmetric-key operations. Therefore we expect the effect of these additional operations to be small while providing significant additional security. Furthermore, the storage integrity checking requires Merkle tree hash operations only on the persistent secure storage tree, not on the whole memory. Similarly, memory replay protection is only necessary on the secure memory lines defined by the TSM and not on the whole memory. This can be accomplished using the Secure Area technique we introduce in Section 5.4.3, or using hardware memory integrity trees \cite{169} that support partial coverage \cite{28}. This results in very significant

\textsuperscript{10}CIC and CEM checking take place at the last level of on-chip cache, which is typically Level 2 or Level 3.
reduction in performance overhead. Furthermore, any overhead incurred is directly related to the level of security protection being provided, and can be customized by the authority or its TSM programmer.

3.9 Summary

We have developed a new architecture for remote trust in portable devices, where a central authority can provide trusted software that will faithfully enforce its policies on securely stored secrets and data. Our authority-mode SP architecture has a very small set of low-complexity hardware features that can be quite easily added to any microprocessor or SOC. We have demonstrated an important example, using this architecture for crisis response where access is critical but security and privacy of sensitive data must be maintained. We provide multiple revocation mechanisms, enabling reliable transient trust of remote users and transient access to this data. Our new persistent secure storage structure, anchored by a root hash (SRH) stored in the processor chip, permits reliable revocation of secrets and associated policies. We demonstrate the new hardware and trusted software mechanisms through a prototype implementation, described in Section 6.6.

A significant contribution of this work is our demonstration that enhanced security can be provided, including remote and transient trust, with only two new hardware registers, the Device Root Key (DRK) and the Storage Root Hash (SRH) registers. Combined with the Concealed Execution Mode and derived keys, the trusted software can use these secrets to provide robust guarantees remotely to the owner of the device (which we call the authority).

While we have used our earlier user-mode SP architecture as a reference design for simple but highly effective processor-based security, we have made many significant new contributions. On one hand, we verified and leveraged some of the architectural features proposed by the user-mode SP architecture, such as the processor features for providing Code Integrity Checking (CIC) and Concealed Execution Mode (CEM) for trusted software. On the other hand, we have provided much stronger trust propositions, solving some open problems in user-mode SP. In particular, our authority-mode SP architecture provides remote trust, transient trust, policy-bound secrets and controlled transitive trust—none of which are achievable with the user-mode SP architecture. All this new secure functionality is possible with only two root secrets, the Device Root Key (DRK) and Storage Root Hash (SRH), without increasing the complexity over the user-mode SP hardware architecture.

This work demonstrates that simple hardware features can be added to provide fundamental hardware anchors that enable more secure systems, without compromising performance, cost or usability. Designing security into the core hardware and software of commodity computing and communications products is a goal of our larger SecureCore collaborative research project [101] and is discussed further in Chapter 6. In the next chapter, we introduce a variant of the authority-mode SP architecture targeted to embedded devices. Later, in Chapter 5, we discuss a new framework for
testing security architectures, which we apply to verify many of the security properties of the authority-mode SP architectures.
Chapter 4

SP on Embedded Devices

4.1 Introduction

The SP architecture is designed as a low-cost addition to a typical microprocessor on a full-fledged personal computer or portable device (e.g., a laptop, netbook, or modern smart phone). Its design assumes the use of large-scale applications and a full operating system, supporting general purpose computing. In this chapter, we present a new design for a simpler version of the SP hardware architecture targeted for embedded and low-power devices, such as consumer electronics or nodes in a sensor network.

These types of embedded devices have different security requirements and different assumptions about attacks. Since the device itself may be single-purpose, there is no need to provide a wide range of security services to other software through complex Trusted Software Modules, as we do in authority-mode and user-mode SP. Instead, many systems need to protect only a small, fixed set of keys and a few basic operations and protocols with those keys. By changing a few of the assumptions that lie beneath the design of authority-mode SP, we can solve many such security problems at a lower cost and on more basic computing platforms. As an example, we closely examine the application of SP to nodes on sensor networks, and in particular the application of key establishment for mobile ad-hoc networks.

Parts of this chapter are taken from [53], which also includes a more in-depth discussion about the mobile ad-hoc networking scenario and key-establishment techniques.1

4.1.1 Sensor Node Platform

Sensor nodes are designed to be particularly low cost and low power devices [181, 84, 81, 107]. Their primary task is usually the collection and dissemination of data from external sensors, used for applications like environmental surveillance and emergency response. Therefore they have a simple OS (e.g., TinyOS [108, 80]), a limited set of

1The study of key-establishment in mobile ad-hoc networking represents joint work with Dahai Xu and the other authors of [53].
software, and no need for interactive user applications (except possibly for installation and debugging). A single SoC (System on Chip) holds the processor core, general purpose memory, as well as EEPROMs to store limited amounts of code and data. Since the SP architecture trusts components on-chip, to the extent they are protected from access by adversaries, we can protect more code and data on-chip in an SoC with a lower cost for cryptographic operations, while still providing the same level of security for an authority-mode Trusted Software Module (TSM).

4.1.2 Wireless sensor networks

Wireless sensor networks, composed of many sensor nodes scattered geographically, build a network between their nodes. In most cases, security is important for protecting and authenticating communication through the network, resulting in the development of a range of key-management schemes [58, 30, 112, 191]. Generally, these schemes place a set of long-term keys into each node, with pairs of nodes following a key-distribution protocol to establish a secure link using these pre-distributed keys. When two nodes want to communicate, they first determine if they share one of the pre-distributed keys. If so, they can use that shared key to communicate securely to generate a new pairwise key for future communications. If they do not share a key, they search for a third node to act as a relay such that each of the original pair of nodes shares a key with the relay node. The relay node is used for the establishment of the pairwise key, after which the pair uses that key to communicate directly.

In mobile networks, where nodes can move and often come into contact (within communication range) with new nodes, there are frequent key-establishment sessions, which provide opportunities for adversaries to eavesdrop and learn the pairwise keys that secure a link. In addition, nodes can be captured, whereupon their key material is copied and can be used to impersonate a legitimate node or otherwise manipulate communications. Furthermore, these captured nodes can be cloned in a variety of ways [190, 127] to create multiple new nodes that make it more likely for an adversary to observe secure communication. In essence, this problem stems from the fact that the long-term keys on a node can be easily extracted when a node is captured, and that the software that uses these keys can be modified to act maliciously within the sensor network. By adding SP protection and a small TSM, we can make it significantly more expensive to extract keys; then we can ensure that the long-term keys are always used according to the correct protocols and only for their intended purposes.

4.2 Embedded SP Architecture

To prevent attacks on the long-term keys of an embedded device, we must tackle the problem of key extraction from a captured node. We first present Reduced Embedded SP, suitable for the simplest devices or sensors. We then extend the solution for slightly more capable devices in the Expanded Embedded SP architecture. These embedded SP architectures provide the same core mechanisms as the regular
authority-mode SP architecture in Chapter 3, but stripped to the bare minimum for devices with very constrained computing and storage resources.

In both versions of the embedded architecture, we are able to simplify the security components by making some new assumptions about the architecture, which do not hold for general purpose computing devices where the full authority-mode architecture is needed.

First, we assume that the embedded device is built as an SoC with some amount of general purpose memory on-chip. A portion of this on-chip memory can be reserved for exclusive use by the TSM, in place of SP’s secure memory operations that encrypt and hash intermediate data stored off-chip.

Second, we assume that the primary secrets being protected by the hardware are long-term secrets, which remain unchanged on the device from initialization through use in the field, and are not updated during operation. This means that the integrity-checking of these secrets can be rooted in on-chip secrets that can only be set during initialization at the authority’s secure depot. Additionally, we assume that it is generally too expensive to perform encryption and decryption on-chip in hardware, and thus allow TSM software to access the device key directly to do these cryptographic operations in protected software. Additional temporary secrets, such as pairwise keys for sensor networks, are stored off-chip; hardware support is not provided for their protection.\(^2\)

Third, we assume that the operating system and applications are less complex, such that interrupts can be disabled during TSM operation. This puts practical limitations on the work the TSM can do, as well as the device’s responsiveness to asynchronous interrupts, but is appropriate for embedded devices. It also means that the standard virtual-memory model cannot be used, at least within the TSM, since page faults cannot be handled during TSM execution.\(^3\) As a result of disabling interrupts, we do not need hardware to handle interrupt protection during CEM (Concealed Execution Mode) while the TSM is running, and the corresponding encryption of general purpose registers can be eliminated, along with the on-chip registers that store the suspended CEM state.

Fourth, since the TSM runs to completion without interference by other software and can execute code from on-chip storage, we can replace the dynamic code integrity checking of the full SP architecture with either on-chip TSM storage or with a hash check performed only upon loading the TSM code.

\(^2\)In sensor networks, the threat model is concerned with attacks on the network as a whole, rather than on single nodes. Thus long-term keys must be protected since their exposure would compromise many nodes and potentially disrupt communication throughout the network. Conversely, the exposure of the pairwise keys of a single node causes damage contained to only communications with that particular node. If some protection is desired, the TSM can use the device key to encrypt and MAC the off-chip data, but the hardware does not provide on-chip writable storage to use as a root of integrity checking.

\(^3\)If any fault or unmaskable interrupt were to occur during TSM execution, it would be necessary for the hardware to zero the general registers and exit CEM before abandoning the TSM thread and returning to the operating system. Thus the TSM will need to be restarted and its internal state in the TSM scratchpad memory may be left in an inconsistent state.
4.2.1 Reduced Hardware Architecture

The simplest version of our architecture, Reduced Embedded SP, is shown in Figure 4.1. It only requires one new register — the device key — and a bit to indicate CEM protected mode. Additionally, the TSM is stored in the on-chip instruction EEPROM and the long-term keys are stored in the on-chip data EEPROM. A portion of the on-chip general-purpose memory of the node is reserved for the TSM Scratchpad Memory.

The TSM is the only software that can use the device key and the protected long-term keys. Since the TSM code is stored within the trusted SoC chip in ROM, it cannot be changed by other software, whether by a malevolent application or a compromised operating system. Similarly, the long-term keys never leave the SoC chip. Any intermediate data, which may leak key bits, generated during TSM execution is placed in the TSM scratchpad memory, and also never leaves the SoC chip.

The TSM code is stored on-chip in a segment of the existing instruction EEPROM along with other system software for the node. Similarly, the long-term keys from the authority are stored in a TSM segment of the data EEPROM. They are encrypted with the device key or with another encryption key derived from it by the TSM in software.
The device key, like the Device Root Key in authority-mode SP, is protected by
the processor hardware; it can only be used by the TSM running in protected mode
and can never be read by any other software.

When the unprotected software wants to make use of protected keys, it calls the
TSM. The TSM functions access the protected keys, perform the requested operation
and return the results, never revealing the keys themselves to the unprotected
software. Each TSM function starts with a Begin_CEM instruction, which disables
interrupts, sets the protected mode bit, and enters protected mode for the next in-
struction. Begin_CEM is only valid for code executing from the instruction EEPROM;
any code executed from the general-purpose memory or off-chip storage cannot enter
protected mode at all. The end of the TSM code is indicated by the End_CEM in-
struction, which clears the mode bit and re-enables interrupts. Table 4.1 shows the
set of instructions used by the TSM and used for initialization in the Embedded SP
architectures.

The TSM Scratchpad Memory is a section of main memory reserved for the ex-
clusive use of the TSM. It is addressed separately from the regular on-chip memory
and accessed only with special Secure_Load and Secure_Store instructions (see Ta-
ble 4.1). These new instructions are available only to the TSM, making it safe for
storing sensitive intermediate data. The TSM can also use this extra space to spill
general registers, to decrypt and store keys, and to encrypt data for storage in regular
unprotected memory. The scratchpad memory is not addressable using normal Load
and Store instructions.\textsuperscript{4}

Initialization of a new device takes place at the authority’s depot. First it must
generate a new random device key. Long-term keys and other secrets encrypted with
it are then stored along with the TSM code on the on-chip EEPROMs. Next it uses
the DeviceKey_Set instruction to store the device key. Finally, any other unprotected
software and data can be copied to the flash storage.

The processor will clear the device key upon writing to either the instruction or
data EEPROM. This ensures that the device key cannot be released to a modified
TSM, that valid data cannot be forged, and that the TSM cannot be changed and
retain access to the encrypted data. Similarly, any time the Device Key register is
set (or cleared), the processor will automatically clear the TSM scratchpad memory,
wiping any intermediate data that was protected by the old key. If in protected mode
at the time, the mode bit is also cleared along with the general purpose registers.

\subsection*{4.2.2 Expanded Hardware Architecture}

The Reduced Embedded SP architecture is ideal for the smallest embedded devices
and sensor nodes that use minimal software and have very limited resources. In
slightly larger devices and sensor nodes with more hardware, the software will be
more complex. The additional applications that run on this sensor, combined with
the TSM and long-term keys, will be too large to store completely on-chip in the

\textsuperscript{4}Alternatively, instead of using Secure_Load and Secure_Store instructions, the TSM scratchpad
memory could be referred by address range only, using normal Load and Store instructions.
SoC. Providing this greater flexibility in the sensor also requires additional support for security. Hence, we propose the Expanded Embedded SP architecture shown in Figure 4.2.

The TSM code and encrypted long-term keys are moved to the off-chip device storage (typically flash memory). This makes them susceptible to modification by other software or through physical attacks. Therefore we must verify their integrity before they can be used. To do this, we add a new register — the Authority Storage Hash (ASH), a hardware hashing engine (implementing SHA-1, MD5, or another cryptographic hash function), a small ROM, and an additional initialization instruction ($ASH\_Set$ in Table 4.1).

The ASH register contains a hash over the entire memory region of the TSM code and long-term keys. It is set by the authority during initialization and is rechecked by the processor each time the TSM is called. The checking code is stored in the on-chip ROM and is fixed and therefore safe from modification; it uses the hardware hashing engine to compute a hash over the TSM code and the encrypted keys.\footnote{Alternatively, the entire hash-checking process can be done in software stored in the on-chip ROM without using a hardware hashing engine to accelerate the hash algorithm.}

When $Begin\_CEM$ is called, the processor disables interrupts and jumps to the TSM-checking routine. If the hash check succeeds, the protected-mode bit is set, and
execution jumps to the newly-verified TSM code, which has access to the verified, encrypted, long-term keys. If the check fails, an exception is triggered.

During initialization, the authority uses the ASH.Set instruction to set the ASH register. This instruction first clears the device key to ensure that the TSM can’t be replaced and still access the protected keys.6

Unlike in authority-mode SP, we allow the TSM to directly read the device key and then trust it to protect the key. This is a more minimal design, eliminating the DeviceKey_Derive instruction (see Table 3.1). Since an embedded TSM is more limited in functionality and the cost of adding hardware is higher, we feel this is a good tradeoff. If desired, a DeviceKey_Derive instruction could be added to the Expanded Embedded SP architecture. It would be implemented as a second routine in the hash checking ROM, taking a nonce parameter and performing a keyed hash with the device key. The DeviceKey_Read instruction would then be removed and a derived key used to encrypt the long-term keys. Since the Reduced Embedded SP architecture does not have hashing in hardware, it must allow the device key to be read by the TSM, which can always implement derived keys in software.

4.3 Key-establishment in Mobile Ad-hoc Sensor Networks

In wireless sensor networks, sensor nodes are deployed, which then set up secure links to their neighbors. A central authority preloads shared secrets into each authorized device in its network, securely before deployment, to enable these links; the links will later be used to pass data through the network and back to a base station and the authority. There are many key-management schemes that have been developed, specifying which secrets are preloaded into each device and what protocol is used to establish links. Our past work [53, 190] studies the security of some of these schemes, specifically looking at the key-establishment process of probabilistic key-management techniques.

In probabilistic techniques, a large pool of symmetric keys is generated by the central authority and each node is assigned some subset of those keys, often randomly, such that two nodes have a certain probability of sharing a key. For example, with a pool of 10,000 keys and 83 keys per node, there is a 50% probability that any two nodes will share at least one key [190]. When two nodes want to establish a link but do not share a key, they must relay through a neighboring node for the key-exchange process.

We consider two primary attack methods on such key-establishment schemes. In a node capture attack, an adversary compromises one or more sensor nodes and extracts their long-term keys after deployment. The adversary then tries to obtain the pairwise

---

6As described in this chapter, the Embedded SP architecture does not provide a Storage Root Hash, as is found in authority-mode SP for the TSM to protect the integrity of data generated during operation. If desired, this could be added back to the embedded architecture, but is not strictly required for the minimal architecture or for the usage scenarios we describe.
Table 4.1: New Embedded SP Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin_CEM</td>
<td>Begins execution of the TSM, entering CEM by setting the Mode bit.</td>
</tr>
<tr>
<td>End_CEM</td>
<td>Ends execution of the TSM, exiting CEM by clearing the Mode bit. Available to TSM only.</td>
</tr>
<tr>
<td>Secure_Store Rd,Rs,imm</td>
<td>Secure store from processor to TSM scratchpad memory. Available to TSM only.</td>
</tr>
<tr>
<td>Secure_Load Rd,Rs,imm</td>
<td>Secure load from TSM scratchpad memory to processor. Available to TSM only.</td>
</tr>
<tr>
<td>DeviceKey_Read</td>
<td>Read the Device Key register. Available to TSM only.</td>
</tr>
<tr>
<td>DeviceKey_Set Rs1,Rs2,sel</td>
<td>Sets the Device Key register, concatenating Rs1 and Rs2 and writing to the selected (sel) double word region of the register. Clears the TSM scratchpad memory, mode bit, and general purpose registers before writing the device key.</td>
</tr>
<tr>
<td>ASH_Set Rs1,Rs2,sel</td>
<td>Sets the ASH register, concatenating Rs1 and Rs2 and writing the selected double word region of the register. Clears the Device Key register before writing the ASH, which in turn clears the TSM scratchpad memory, mode bit, and general purpose registers. Available in the Expanded Embedded SP architecture only.</td>
</tr>
</tbody>
</table>

keys used by other nodes so that it can later monitor their links. If it shares the long-term key used by two nodes for key-establishment, it can observe the corresponding negotiated pairwise key between those two nodes. Alternatively, if the two nodes do not share a long-term key, they may choose to relay through the compromised node, which can save the resulting pairwise key. In both cases, the adversary is limited to attacking nodes within its communication range.

In a node fabrication attack, the adversary uses the extracted keys to fabricate new nodes. One method is to simply clone the compromised node, using additional sensor devices loaded with an exact copy of the keys from the compromised node. Another method pools the keys from multiple compromised nodes; it then either makes fabricated nodes [127] with unique subsets of the combined key pool or fabricates super-nodes using all of the extracted keys in each copy. Cloning allows adversaries to extend their communication range, and node fabrication increases the probability that the adversary’s compromised node will be chosen as a relay. Both attack methods allow adversaries to significantly increase their chance of successfully attacking a pairwise link compared to a node capture attack alone [190].

The crucial observation is that the attacks succeed because the long-term keys are not protected when a node is captured. We assume an adversary with physical access
to the device, so software protections are easily bypassed. The keys are accessible
to the software on the node, which the adversary can exploit or replace entirely.
The adversary might also read the keys directly from a flash memory chip or other
permanent storage when the device is offline.

In our solution, we protect these secrets with SP, such that the device key is
used to encrypt the long-term keys, making them accessible only to the TSM, which
cannot be modified without erasing the device key. Thus the long-term keys are never
stored unencrypted on the permanent storage, and the adversary cannot access the
keys through software or physical attacks. As is also the case with the other SP
architectures, the security is rooted in the protected on-chip secrets — the hardware
roots of trust.

4.4 Security Analysis

4.4.1 Attacks on Protected Keys

Our new Embedded SP architectures safeguard a device’s long-term keys, preventing
extraction by an adversary in the event of capture. The keys are always stored in
encrypted form in permanent storage in either on-chip EEPROM or off-chip stor-
age. The adversary cannot obtain the device key needed to decrypt them, since it
is protected on-chip; the device key never leaves the SP processor or its protected
software environment. Therefore, as in authority-mode SP, rather than access the
keys directly, regular software must call TSM functions that perform operations with
the keys on its behalf. Thus software can use the keys in any way permitted by the
TSM, but can never extract the keys themselves in plaintext, even under physical
attack.

Node Fabrication Attacks

Without SP protection, adversaries maximize their attack success by cloning multiple
copies of compromised nodes and combining the nodes’ long-term keys. This increases
their ability to observe link establishment and the likelihood of being used as a relay
node. With SP protection, they cannot create any clones and are limited to using
only the keys originally stored on each captured node individually.

Node Capture Attacks

Node capture attacks use long-term keys in the node to observe pairwise links be-
tween other nodes in the network. With SP, adversaries can no longer extract the
keys. However, they can still change unprotected software that calls the TSM. A
simple TSM might provide functions like $\text{Encrypt}(\text{key}, \text{data})$ and $\text{Decrypt}(\text{key}, \text{data})$.
Adversaries can use the keys through this TSM interface to observe or attack pair-
wise links without ever seeing the actual keys. While we do not prevent node capture
attacks outright, such attacks are limited since the adversaries can only observe links
within the communication range of the compromised node. This severely limits the
fraction of the network that they can attack with each captured node. Simulation results [53] show that this restriction reduces the severity of a node capture attack from 9.7% of the network without SP protection to 2.1% of the network with SP protection when only 3% of nodes are captured. This is much less than the fraction of the network that can be attacked via node fabrication attacks [53, 190], which the same simulation results show to reach 42.6% of the network if six copies are made of each captured node.

Eavesdropping Attacks

A node’s temporary and pairwise keys can also be encrypted with the device key and protected by the TSM, preventing extraction of these keys and ensuring that proper protocols are followed. The TSM functions can then be modified to restrict which keys can be used with Decrypt(). If all long-term keys and pairwise keys cannot be used with Decrypt(), then adversaries who can call the TSM will not be able to use those keys to eavesdrop on network communications.

Instead, nodes can include new functions such as Relay(key1, key2, data). This function allows a node to act as a secure relay node, i.e., to decrypt data from one link with key 1 and re-encrypt it in the TSM with key 2 to send out another link, without ever revealing the plaintext data outside of the TSM to an adversary that might control the physical device.

Other TSM functions can prevent the adversary from observing the key exchange or pairwise communication of nodes within its transmission range, or other sensitive operations. In this way, a node could continue to operate securely despite being captured by an adversary, who can now only perform a denial of service attack by disconnecting the node from the network, either before or during communications. Adversaries cannot eavesdrop on links for which they are a relay node, unless they otherwise have a copy of the pairwise key.

If the temporary and pairwise keys are encrypted and hashed, they are still susceptible to replay attacks. The ASH only covers the TSM code and long-term keys, which are static and cannot be changed during operation. The ASH can only be updated during the initialization process by the authority. Therefore, if any of these keys might be changed or revoked, for example because a node is removed from the network, then it is necessary to prevent such attacks. To accomplish this, a Storage Root Hash register can be added to the Embedded SP architecture to provide an on-chip root secret for integrity that can be updated during operation. As in authority-mode SP, an SRH register would be accessible to TSM code and allows revocation and policy changes to be made permanent.

4.4.2 Attacks on Changing the TSM or the Device Key

The security of the long-term keys relies on the correctness and proper design of the authority’s TSM. As part of the trusted computing base of the system, this software must not leak the secrets that it can access directly. This includes preventing leaks through any intermediate data written to general purpose memory, off-chip storage,
or left in general registers when it exits. The TSM runs with interrupts disabled, so no other software will have an opportunity to observe its registers or modify its code or data while it is executing. If the TSM ever exits abnormally due to an exception, the processor clears the general registers before ending protected mode. Any other sensitive data will be in the TSM scratchpad memory, which other software cannot access.

In order to circumvent the access control provided by the authority’s TSM, attackers might try to replace it with their own TSM or modify the existing TSM. In the Reduced Embedded SP architecture, the TSM and long-term keys are stored in on-chip EEPROM where they cannot be modified without clearing the device key and all intermediate data. Once the device key is cleared, the attacker’s TSM will not have access to the key required to decrypt any long-term keys that remain in the EEPROM. In the Expanded Embedded SP architecture, the attacker could modify or replace the TSM code in off-chip storage. The hash checking routine will detect any such modifications made to the TSM before execution. Since data in off-chip storage could be modified through a physical attack during execution, the TSM and long-term keys should first be copied to general purpose memory on-chip before being verified and executed, where they will be safe from physical attacks.

Next, if the attackers try to modify the ASH register to match the new TSM code, the device key will be cleared, irrevocably cutting off their access to all of the keys that were encrypted with that device key. Clearing or setting the device key also clears the TSM scratchpad memory, so any intermediate data stored there that might have leaked secrets is also unavailable to the new TSM. Correspondingly, the code in the hash-checking ROM is assumed to be read-only, however if it can be reprogrammed, then doing so should also clear the ASH, device key, and TSM scratchpad memory.

Finally, if attackers choose to change the device key without changing the TSM, there is no concern since the long-term keys will still be encrypted under the old device key and will therefore become inaccessible to an attacker. Any intermediate data in the TSM scratchpad memory or in general registers that might have revealed the long-term keys is also cleared when the device key is changed.

The TSM code itself is not secret and may be used freely by attackers. Without access to the original device key and the encrypted long-term keys, the attackers will be unable to authenticate to any of the other legitimate nodes in the network, serve as a relay node, or otherwise attack the network, even though they have the TSM code and the correct network protocols. At most, attackers could use the TSM code and any stolen nodes to setup their own sensor network that would operate independently of the network setup by the authority.

4.5 Summary

In this chapter, we show how the SP architectural concepts can be applied to embedded devices, where hardware resources are severely limited, while still providing strong security guarantees for protected secrets. We offer two variations of the Embedded SP Architecture to fit a variety of devices, and discuss how these would be
used to secure nodes in wireless sensor networks. We analyze node capture and node fabrication attacks on sensor nodes and show how to use Embedded SP to protect the key-establishment process against these attacks.

We have offered an analysis of the security of the hardware and software mechanisms involved in Embedded SP. Additionally, [53] presents simulation results for how the resilience to node capture and node fabrication attacks improves the overall security of mobile ad-hoc sensor networks.

Regarding cost, our architecture is easily integrated into the SoC of standard embedded processors. Even so, to be the most practical for embedded devices, the hardware requirements have been significantly reduced from the original authority-mode SP. We have eliminated the hardware for the encryption/decryption engine, the hardware and registers for protecting CEM state during interrupts, and the cache tag bits. We have also simplified the hardware for secure memory — it now only addresses a restricted segment of the on-chip memory rather than using cache tags and encryption/decryption on L2-cache evictions and misses. And finally, we have simplified the integrity checking hardware for the TSM — from dynamic code integrity checking (using MAC verification) to a ROM-based software hash verification routine upon TSM function calls for the Expanded Embedded SP architecture, and to a simple check of the code location (from the on-chip instruction EEPROM) for the Reduced Embedded SP architecture.

In addition to reducing the hardware cost, our simpler design for Embedded SP helps keep down power requirements by limiting the amount of cryptographic operations that need to be performed in order to secure the storage and use of long-term keys. Especially on sensor nodes, which must operate in rough conditions with limited power sources, security hardware and software must not consume too much energy [27, 183].

In summary, the Embedded SP Architecture provides robust security at low cost for a wide range of devices, as described for preventing the extraction of long term keys in sensor nodes. This adds to the validation of our general SP architecture by demonstrating its applicability to another platform, in addition to user-mode SP and authority-mode SP for general purpose processors. We further validate the SP architecture in the next chapter, by testing the security properties of the architecture and the design of trusted software.
Chapter 5

A Framework for Testing Hardware-Software Security Architectures

5.1 Introduction

Validating the design of the authority-mode SP architecture (Chapter 3) is a complicated endeavor. While an in-depth analysis of its security features has been performed, the full hardware-software architecture must be implemented and tested to verify that the security mechanisms are sufficient and provide the required security properties to defend against adversaries. The usefulness of the SP architecture further depends on demonstrating that security-critical applications can be built effectively on such an architecture.

More generally, designers of security architectures face this challenge whenever it is necessary to place low-level security mechanisms in the hardware or operating system kernel that higher-level software layers rely upon to support security-critical applications. The resulting architecture is a combination of hardware, kernel, and application software components that are difficult to test together. The security of the system as a whole relies on both the correct design and implementation of the low-level security features, the correct and secure use of those features by the software layers, and the security of the software components themselves (e.g., their resilience to attacks). Therefore, we need a framework that can comprehensively model the architecture and study the interactions between hardware and software components. Furthermore, the security mechanisms at each level of the system must be robust under normal operation as well as when under active attack by an adversary. We propose a testing framework that emulates the hardware components of a security architecture and provides a controlled environment with a full software stack, with which coordinated security attacks can be performed and observed.

Our framework provides a platform for penetration testing of both the hardware components and the full software environment. Faults can be injected to emulate postulated attacks and to test the effectiveness of proposed countermeasures. The
framework can also be used to verify that code behavior conforms to known best practices and to detect known vulnerabilities. While it can model attacks using unknown penetration methods by instead modeling the impacts of of these attacks, it cannot however simulate clever adversaries who may devise entirely new attack methods.

We have designed our testing framework [50] with the goal of verifying the SP architecture [51, 102], while being generalizable to other security architectures. Testing SP requires that the framework models attack methods, using known and unknown penetration methods, and enables studies of their impact on the security mechanisms provided by the hardware and software. The testing environment — including the hardware implementation, software stack, threat models, and attack mechanisms — must be as realistic as possible so that results are meaningful. As far as we know, no existing testing methods provide the ability to mount coordinated attacks on new integrated hardware-software security architectures and observe their effects, as our framework does.

Furthermore, our framework allows testing to be done during the design time; this gives confidence in the architecture before the complete system is built, at which point it is costly to make fundamental changes in response to security flaws. While we focus our analysis on SP, a hardware-software security architecture, our framework is equally useful for testing software-only architectures.

Our Approach

Our testing framework is composed of two components: a system under test (SUT) containing the new hardware architecture and software stack under consideration, and a testing system (TS) that coordinates monitoring of, and attacks on, the SUT. The SUT is attacked by the TS according to the threat model being tested; the TS itself is not attacked. The TS monitors hardware and software events that occur in the SUT using hooks provided by our framework. It also can inject attacks at all layers of the system. An attack script running in the TS coordinates events and attacks from both hardware and software components in the SUT.

Our testing framework can model real attack mechanisms using known penetration mechanisms. It can also model unknown future attacks by more powerful adversaries by enabling direct attacks on software and hardware components that go beyond known penetration methods. We do this by mapping attacks to their impacts on the SUT. Hence, a key strength of our system is that it allows design-time testing assuming a very powerful attacker to test the limits of the SUT, without the need to find a specific penetration path through the system.

The primary contributions of this work are:

- a new flexible framework for design-time testing of the security properties of new hardware-software architectures;
- execution of a realistic software stack, using commodity operating systems, for testing different applications that use the new security mechanisms;
• a flexible, fast, and low-cost method for emulating hardware security features, using virtualization, for the purpose of design validation — without the need for costly and time-consuming fabrication of hardware prototypes;
• the ability to simulate the impact of very powerful attackers for “future” attacks;
• an improved architecture for SP’s secure memory mechanism and its implementation; and
• the application of our framework toward the validation of the security properties of the SP architecture, by providing a suite of attacks on SP’s trust chain.

Significant portions of this chapter have been documented in previous technical reports [50, 49, 32] and will be submitted for publication.¹

5.2 Threat Model and Assumptions

For this work, we focus on hardware-software architectures where new hardware security mechanisms are added to a general-purpose computing platform to protect security-critical software and its critical data. The hardware provides strong non-circumventable security protection, and the software provides flexibility to implement different security policies for specific applications and usage scenarios.

Figure 5.1 shows a typical system with a trusted software application and the addition of new trusted hardware security mechanisms to the CPU (e.g., new instructions, registers, exceptions, and hardware mechanisms). Sometimes, as shown in the figure, the OS cannot be trusted, especially if it is a large monolithic OS like Windows or Linux. Other times, an architecture might trust parts of the operating system kernel (e.g., a microkernel [110, 78]), but not the entire operating system.

The figure also shows the three classes of attacks that we consider in our testing framework. First, malware or exploitable software vulnerabilities that can allow adversaries to take full control over the operating system to perform software attacks. They can then access and modify all OS-level abstractions such as processes, virtual memory and virtual memory translations, file systems, system calls, kernel data structures, interrupt behavior, general registers, and I/O.

Second, hardware attacks, which can be performed by adversaries with physical possession of a device, such as directly accessing data on the hard disk, probing physical memory, and intercepting data on the display and I/O buses. For software-only security architectures, we can also model some software attacks as having the same impact as these physical attacks.

Third, network attacks that can be performed with either software or hardware access to the device, or with access the network itself. Some network attack mechanisms act like software attacks (e.g., remote exploits on software), while others attack the network itself (e.g., eavesdropping attacks) or application-specific network protocols (e.g., modification attacks and man-in-the-middle attacks).

¹The testing framework was designed and implemented as joint work with Mahadevan Gomathisankaran.
In order to adequately test a new security architecture, all of these attack mechanisms must be considered and tested, according to the threat model of the particular architecture. Our testing framework provides hooks into each relevant system component, and allows information and events at each level to be correlated to emulate the most knowledgable attacker.

Overall, we consider the functional correctness of the new hardware security mechanisms and the security-critical software components, as well as the interaction between these hardware and software components. We do not consider timing or other side-channel attacks.

Our threat model thus far describes the range of attacks that our framework must test in the SUT, and which system components of the SUT should be observable and controllable by the TS. The framework itself, however, is completely trusted as a simulation environment and is not itself attacked. No real sensitive information is stored or used in the testing framework. Therefore it is safe for the TS to have access to all low-level data of the SUT’s hardware and software, even when this would violate...
the assumed security guarantees of the emulated system (e.g., by reading/writing secrets from on-chip SP registers).

As a testing environment, we additionally consider the implementation of the components of the security architecture. We write attacks and run tests on the implementation of the emulated hardware components and the design of the hardware interfaces. Problems discovered while writing hardware emulation routines may shed light on flaws likely to occur when building real hardware. Thorough testing of hardware interfaces may reveal vulnerable combinations of hardware state and inputs that were not carefully considered in the original design. This testing is critical to the security of the real system when it is manufactured and deployed.

In the design of an architecture like SP, it is convenient, if not necessary, to overlook buggy or malicious hardware as an orthogonal problem within the manufacturing process.\(^2\) We do not attempt to solve this problem — SP assumes hardware to be non-bypassable and also correct according to its design. That said, to the extent that the emulated system corresponds functionally to the real microarchitecture, the framework can be used to generate data for test cases to run against manufactured devices, or to provide inputs to other verification schemes.

### 5.3 Testing Framework

The design goal of our testing framework is to create a generic platform that can emulate and test a wide range of security architectures in a realistic environment. It must test the new security hardware and the architecture-specific application software that uses this hardware, running on top of a full commodity operating system. The testing framework should allow easy monitoring of software and hardware events, and allow modification of software and hardware state to mimic attacks on the system.

#### 5.3.1 Architecture

We build our testing framework on top of existing virtualization technology, which allows us to run a full set of commodity software efficiently. A virtual machine monitor (VMM) is the software that creates and isolates Virtual Machines (VMs), efficiently providing an execution environment in each VM which is almost identical to the original machine [140, 145]. By modifying an existing VMM’s hardware virtualization, we can augment the virtual machine to have the additional hardware features of a new security architecture in addition to those of the base machine. Using virtualization allows the unmodified hardware and software components to run at near-native speed, while permitting our framework to intercept events and system state as needed.

---

\(^2\)Hardware bugs could violate the security properties of SP in many ways. For example, mistakes in the encryption and hashing engine algorithms could weaken the cryptography; faulty access control checks could reveal protected keys or derived keys; errors in suspending or resuming CEM could reveal intermediate data in registers or secure memory; and improper CIC checking could cause TSM code to be modified.
Our Testing Framework is divided into two systems, as shown in Figure 5.2, the System Under Test (SUT) and the Testing System (TS), each running as a virtual machine on our modified VMM. The SUT is meant to behave as closely as possible to a real system which has the new security architecture. It must behave as if it has all of the new security primitives available in hardware, along with the corresponding protected software for that architecture. In our current system, the SUT runs a full commodity operating system (Linux) as its guest OS, which is vulnerable to attack and is untrusted.

Figure 5.2: Testing Framework Design

The TS machine simulates the attacker, who is trying to violate the security properties of the SUT. It is kept as a separate virtual machine so that the TS Controller can be outside of the SUT to launch hardware attacks. The virtualization isolates all testing activity and networking from the host machine.

The testing framework itself is independent of the threat model of the system being tested, and hence enables full controllability and observation of the SUT in both hardware and software. This makes it suitable for many purposes during the design phase of a new architecture. During the initial design and implementation of the system, the TS can act as a debugger, able to see the low level behaviors in hardware, all code behavior, and data in the software stack. When testing the supposedly correct system, the TS is the attacker, constrained by a threat model to certain attack vectors.

A particular point of elegance of our framework is that the threat model can be easily changed, and the set of attack tools given to the attacker adjusted for
each test. The framework can be used for any combination of mechanisms: access to internal CPU state of the virtual processor, “physical” attacks on the virtual machine hardware (e.g., hardware probes on the buses, memory, or disk), software attacks on the operating system (e.g., malware installed in the OS kernel), and network attacks (e.g., interception and modification of network packets and abuse of network protocols and application data). For example, in some cases, it might be desirable to perform black-box testing of a new design using only the network to gain access to the SUT, while in other cases, white-box testing will allow the attacker knowledge about the system’s activities, such as precise timing of attacks with hardware interrupts and breakpoints into the application code, or observation of data structures in memory.

5.3.2 Testing Framework Components

The main components of our Testing Framework are shown in Figure 5.2, depicting the System Under Test and the Testing System. The framework detects events in the SUT and provides access to the full system state to the TS, using both hardware and software channels to communicate. The TS Controller (TSC), running on the TS, is the aggregation point that receives events from both channels. It receives OS and Application level (software) events from the SUT via a network channel and receives hardware events from the VMM. These same channels are used to change the system state of the SUT at the hardware and software layers.

The TS Proxy (TSP) is added to the SUT to communicate with the TS Controller to receive commands and send events back. It simulates the effect of a compromised operating system for launching software attacks, allowing the OS to be fully controllable by the TS. The TS Controller and TS Proxy are each divided into user-level and kernel-level components. Additional trusted entities of the security architecture that are not under test, such as network servers, may be hosted in the TS and report their activity directly to the TS Controller.

Attack Scripts specify how particular attacks are executed on the SUT, making use of the TS Controller through an API. The Controller passes hardware and software events to an attack script, which can respond by accessing and modifying the state of the SUT. The script monitors events and dynamically responds to them in order to successfully launch coordinated attacks, or to detect that an attack was prevented by the security architecture. The TS Controller API is described in detail later in Section 5.3.3; additional examples of attacks are given in Section 5.5, including a sample attack script. For debugging purposes, simple scripts can be written to provide command-line access to the SUT state, or access via the TS Controller API can be integrated into other software.

To capture events and access system state in the SUT, the modified VMM monitors and controls the hardware with the Event & Attack Module, making hooks into the virtual CPU and virtual devices as well as into the new Security Hardware Emulator for new hardware not present in the base CPU. When properly instrumented, the VMM is able to intercept and capture hardware events such as interrupts/faults, breakpoints, memory accesses, and even the execution of individual instructions. It can also directly access CPU registers, physical memory, and virtual memory map-
pings in the page table. The Security Hardware Emulator implements the new SP features and already provides hooks for new SP instructions, faults, and access to new registers and protection mechanisms (e.g., interrupt protection and code integrity checking). We later describe the event handling process in detail in Section 5.3.4.

The TS Proxy monitors and controls the applications and OS directly from within the SUT. Its kernel module has direct access to kernel data structures and can intercept system calls, modify process scheduling, generate signals to other processes, access virtual memory and memory mappings, access file systems, etc. The TS Proxy also is responsible for launching and controlling the protected application. This application is the actual implementation of the software that would be run on the secure architecture. It should demonstrate the features of the architecture and possibly interact with a local user to provide access to secrets and data that the architecture is protecting. The application may also be instrumented to report application-level events, using system calls to report its behavior to the TS Proxy and in turn to the TS Controller. Alternatively, the TS Proxy and the VMM can monitor its behavior indirectly through other mechanisms, without modifying the application code.

Communication of events and data between the SUT and TS occurs asynchronously, through a network channel for software events/attacks from the TS Proxy and through a custom inter-VM channel within the VMM (implemented over shared memory) for hardware events/attacks. When synchronization is necessary, either the application or the entire SUT machine can be frozen to preserve state, while the TS and attack scripts continue to execute; thus the TS can launch attacks or make observations with precision relative to particular instructions or system events. When the entire SUT is frozen, the hardware channel is still functional and can access/modify state, while the TS Proxy cannot run to provide the network channel.

Within the virtual machines, the components communicate through a combination of new system calls (to kernel components), hyper-calls (direct to the VMM), signals (from kernel to user components), and virtual hardware interrupts (from the VMM to kernel components).

By using two separate virtual machines to host the components of our testing framework, we are able to keep the SUT as close as possible to the real security architecture being tested, including all of its software components, both trusted and untrusted. We do not need to trust the SUT’s guest OS for the sake of correctness of the emulation of new hardware security features. In fact, the SUT’s guest OS can launch real attacks on the system (as we have instrumented it to do through the TS Proxy, as controlled by the TS).

### 5.3.3 Events and Attacks

Table 5.1 lists various events and attacks that can be exposed by the framework for each layer of the system. The hardware layer is further classified into events and attacks from the base hardware (x86 architecture in our work) and the new emulated security architecture.
<table>
<thead>
<tr>
<th>Layer</th>
<th>Events Monitored</th>
<th>Impact of Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protected Application</td>
<td>API function entry/exit, Library calls, User authentication, Network messages, Other application-specific events.</td>
<td>Read/write application data structures, Trigger application API calls, Intercept/modify network messages, Other application-specific attacks.</td>
</tr>
<tr>
<td>OS</td>
<td>Memory access watchpoints, Virtual memory paging, File system access, System calls, Process scheduling, Instruction breakpoints, Device driver access, Network socket access, Interrupt handler invocation, etc.</td>
<td>Read/write virtual memory, Read/write kernel data structures, Read/write file system, Intercept/modify syscall parameters or return values, Read/write suspended process state, Modify process scheduling, Intercept/modify network data, Modify virtual memory translations.</td>
</tr>
<tr>
<td>Base Hardware (x86)</td>
<td>Privileged instruction execution, Triggering of page faults and other interrupts, Execution of an instruction pointer.</td>
<td>Read/write general registers, Read/write physical memory, Trigger interrupts, Intercept device I/O (e.g., raw network &amp; disk accesses).</td>
</tr>
<tr>
<td>Secure Hardware</td>
<td>Execution of new instructions, Triggering of new faults, Accesses to new registers.</td>
<td>Read/write new registers &amp; state, Read/write protected memory plaintext.</td>
</tr>
</tbody>
</table>

Hardware events are monitored through the VMM hooks during execution and can be as fine-grained as the execution of a single instruction or HW operation in the SUT. The VMM freezes the SUT as it communicates each event over the inter-VM channel, allowing the TS to possibly change the result of that operation before it completes. Software events and attacks rely on hooks from the TS Proxy into the OS kernel through its kernel module, and to the testing application using its user-mode component.\(^3\) The Proxy can also function as a debugger tool, reading the application’s memory and accessing its symbol table to map variable and function names to virtual addresses. The application can optionally be instrumented to access its state and events. The TS Proxy suspends scheduling of the application’s process (along with its TSM for the SP architecture) while it is communicating an event to the TS Controller or performing an attack.

\(^3\)Many of the software events and attacks can also be implemented through the VMM. For example, system calls can be monitored by watching for software trap instructions, with parameters and results accessed for reading and writing through access to physical or virtual memory.
Table 5.2 lists the API that the TS Controller exports to the attack scripts, combining the hardware and software channels. First are commands used to launch and control the execution of the application under test on the SUT. The second group of commands control event handling\(^4\), and the last group provides access to SUT state.

As an example, if an attack script wanted to modify network traffic sent to an application, it first must capture the network data. The application receives data via a `read()` system call; this can be intercepted at the hardware level as an interrupt, checking the parameters in the registers for the syscall number, or can be intercepted by the TS Proxy within the syscall handler of the OS. Alternatively, the virtual network card could be instrumented to send network events to the TS. Once the TS receives the `read()` event, it can read the buffered network data from memory, and possibly modify the data to launch its attack before resuming the SUT and protected application. The attack script would then proceed to monitor other events to observe the effects of the attack, such as the application updating its keys on disk or modifying state in the security architecture. Such an attack is readily implemented using the TS Controller API in Table 5.2 and the events in Table 5.1.

As shown in this example, the attack mechanisms we provide are designed around the impact of attacks on the SUT’s state. In this case, it was not necessary to specify how an attacker would intercept network traffic. This is preferred since (1) new attack penetration methods are frequently discovered after a system is deployed and often are not foreseen by the designer, (2) most real attacks result in or can be modeled by the impact of attacks which we provide in Table 5.1, and (3) the attack scripts themselves can be restricted to model specific penetration methods when testing for a more limited attacker.

Table 5.3 provides examples of how common penetration methods map to impacts on the SUT state that we can model in the framework. We then specify what attack methods in the testing framework would produce the same effects. We also specify which features of the SP architecture might protect a TSM from being vulnerable to the attack.

For example, when an operating system is compromised with a rootkit, an attacker can manipulate the execution of an application, can read/write the application’s virtual memory, and can change the results of any system calls made by the application. In the testing framework, the VMM can read and write any address in virtual memory while pausing execution, and can intercept interrupts made for system calls. Alternatively, the TS Proxy can modify kernel data structures to affect process scheduling or to access the applications memory space; it can change interrupt tables to change system call behavior, adding wrappers to any functions that it would like to modify. The SP defenses rely on CEM protection of the application — secure memory prevents unauthorized observation and modification of data, and code integrity checking prevents modifications to the control flow of the application. The TSM can also validate the results of system calls, and this checking cannot be bypassed by modifying its code or return addresses.

\(^4\)The watch list can wait for any of the event types in Table 5.1. Event parameters and data are either passed to the TS directly or are accessible via pointers to the SUT’s memory with `Access_Mem`. 

75
Table 5.2: TS Controller API for Attack Scripts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h ← INIT()</td>
<td>Initialize the Controller and return a handle h to access resources.</td>
</tr>
<tr>
<td>EXECUTE(h, app, params)</td>
<td>Execute the application app on the SUT with the given parameters params.</td>
</tr>
<tr>
<td>INTERRUPT(h, num)</td>
<td>Trigger an immediate virtual hardware interrupt number num on the SUT.</td>
</tr>
<tr>
<td>BREAKPOINT(h, addr)</td>
<td>Setup a breakpoint to interrupt the SUT at an address addr.</td>
</tr>
<tr>
<td>EVENTADD(h, eventType)</td>
<td>Add the eventType to watch-list.</td>
</tr>
<tr>
<td>EVENTDEL(h, eventType)</td>
<td>Delete the eventType from the watch-list.</td>
</tr>
<tr>
<td>event ← WAIT(h)</td>
<td>Blocking call that waits for any event in the watch-list to occur in the SUT. Once an event is triggered, the SUT is paused and the TS continues running the attack script. An application exit in the SUT always causes a return from WAIT().</td>
</tr>
<tr>
<td>event ← WAITFOR(h, eventType)</td>
<td>Similar to WAIT() but waits for the specified event (or application exit), regardless of the watch-list.</td>
</tr>
<tr>
<td>CONT(h)</td>
<td>Execution of the SUT is resumed, after an event or interrupt.</td>
</tr>
<tr>
<td>ACCESS_GENREG(h, r/w, buf)</td>
<td>Reads/writes (r/w) the general registers or SP registers of the SUT to/from buf.</td>
</tr>
<tr>
<td>ACCESS_SPREG(h, r/w, buf)</td>
<td>Reads/writes (r/w) the SP registers of the SUT to/from buf.</td>
</tr>
<tr>
<td>ACCESS_MEM(h, v/p, r/w, addr, sz, buf)</td>
<td>Reads/writes (r/w) sz bytes from virtual or physical memory (v/p) of the SUT at address addr to/from the buffer buf. Can access memory regularly or as an SP secure region (accessing the plaintext of encrypted memory).</td>
</tr>
</tbody>
</table>

For side channel attacks, an attacker might monitor power lines or emissions from the device to indirectly observe memory contents. Since these are noisy channels, such accesses are not perfect and can contain errors. Similarly, attackers can adjust the supply voltage to a chip to introduce errors in memory, which can sometimes be exploited to reveal keys. We can model these attacks in the framework by having each read or write succeed only with some probability, introducing noise intentionally. SP defenses for side channel attacks vary depending on what tamper resistance is added to the chip. Secure memory may cause some attacks to only reveal encrypted data,
Table 5.3: Mapping Attacks to Testing Framework Modules

<table>
<thead>
<tr>
<th>Penetration Method</th>
<th>Impact</th>
<th>TF Attack</th>
<th>SP Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer overflow</td>
<td>Hijack procedure return, modify function pointer, return to libc, etc.</td>
<td>Intercept</td>
<td>Code integrity checking</td>
</tr>
<tr>
<td>Format string attack</td>
<td>Modify arbitrary memory address</td>
<td>Read/write virtual memory</td>
<td>Secure memory</td>
</tr>
<tr>
<td>Virus/malware</td>
<td>Access filesystem, send network traffic, access or intercept user I/O</td>
<td>Intercept system calls or virtual devices</td>
<td>TSM secure storage</td>
</tr>
<tr>
<td>OS rootkit or malicious device driver</td>
<td>Full control over OS (all OS and Application attacks)</td>
<td>Intercept system calls, read/write kernel data, modify scheduling, etc.</td>
<td>Protected mode for TSM</td>
</tr>
<tr>
<td>Memory probe (e.g., cold-boot attacks on memory [75])</td>
<td>Arbitrary access to physical memory</td>
<td>Freeze SUT, read/write physical memory</td>
<td>Secure memory</td>
</tr>
<tr>
<td>Side channel attacks</td>
<td>Probabilistic access to cache, memory, etc.</td>
<td>Read/write physical memory with random chance of success</td>
<td>Secure memory, etc</td>
</tr>
</tbody>
</table>

and the chip may detect attacks and wipe clear some of the on-chip keys or secure cache lines to prevent observation.

5.3.4 Implementation

We implemented a prototype of our testing framework on VMware’s virtualization platform [182], including all of the components in Figure 5.2, and some events and attacks at each system layer.

Through the VMM, we have implemented events and attacks for the SP hardware (execution of new instructions, new faults, and read/write access to new registers) and for the base hardware (capture of selected interrupts, raising of hardware interrupts, and access to virtual memory, physical memory, and general registers). Through the TS Proxy, the implemented events and attacks include access to all system calls, process scheduling of the TSM application, virtual memory translations, and any application events that are instrumented in the application code.
The Security Hardware Emulator, VMM Event & Attack Module, and inter-VM communication channel required modifying the source code of the VMware VMM. The kernel components of the TS Proxy and TS Controller are implemented as Linux kernel modules. The TS Proxy application is implemented as a Linux user process and controls the execution of the Testing Application.

The TS Controller application is implemented as a static library which is called by the Attack Scripts, which are currently implemented as compiled programs rather than interpreted scripts. Upon initialization, the TS Controller connects with the TS Proxy over the virtual network. The TS Proxy and TS Controller each register with their respective Event & Attack Module via a hypercall to the VMM. The Controller API then allows the script to launch applications on the SUT (such as the protected testing application), to monitor events, and to modify state to implement attacks.

The Security Hardware Emulator emulates the SP architecture, as described in Section 5.4.2, including its hardware roots of trust, secure memory, and interrupt protection. We have also implemented a library of protected software for SP, which is used for a remote key-management application as described in Section 5.4.5. Our testing application uses this library to exercise the software, and in turn, the SP hardware.

Hardware Event Handling

Figure 5.3 shows the steps taken by the testing framework to detect an event, in our current implementation. In step 1, the protected software (TSM) makes a call to one of the new SP instructions (e.g., to generate a new cryptographic key). This requires making a hypercall down to the SP Hardware Emulator Module to emulate the instruction and produce results into the general registers. In step 2, the Emulator Module notifies the Event & Attack Module on the SUT machine of this hardware event (the execution of an SP instruction). The event is then sent over the inter-VM communication channel to the Event & Attack Module in the TS machine. In step 3, the event is passed up to the Controller kernel module via a virtual hardware interrupt/IRQ, and the VMM freezes execution of the entire SUT. In step 4, the Controller application is notified of the event via a Linux signal. The Controller makes a hypercall (Get_Event() — see Table 5.4) to the VMM to retrieve the details of the hardware event. In step 5, the Controller returns the event to the attack script which was waiting for notification of this event (e.g., key generation instructions). It can now continue executing the script to perform any hardware attacks while the SUT is frozen and then send a command (CONT()) to the VMM to resume the SUT and continue execution. These steps are repeated for every hardware event monitored by the Controller.

Communication Channels

The framework uses two communication channels between the SUT and TS to pass data to observe and modify system state. The software channel is used primarily for high-level OS and application events. When the TS Proxy captures an event from
the TSM or its application, it suspends the process so that the event can be handled by the TS before any additional code executes in the testing application. When it captures an event from the OS kernel, it is not necessary to suspend the testing application’s process since that process was not executing at the time. The TS Proxy waits for a response from the TS as each event occurs before resuming the application. Since the TS Proxy can suspend processes appropriately to ensure no relevant code executes before the response, there is no need to suspend the entire SUT machine for software events. For communications, the TS Proxy acts like a server and accepts a standard network connection from the TS Controller. While handling an event, the TS Proxy makes calls into its kernel module to access the state of the OS and also makes hypercalls directly to the VMM if it also wishes to be notified of hardware events or wishes to create hooks to instrument additional hardware events.

The hardware channel connects the Event & Attack Modules of each VM within the Virtual Machine Monitor and is shown in detail in Figure 5.4. Due to the design of VMware’s VMM, each VM is mostly isolated such that the internal state of one VM is not accessible to the other. Furthermore, each VM is implemented as multiple components, with a “VMX” process for each virtual CPU running in the context of the host OS, and the “monitor” (the VMM), which runs on the bare processor hardware without OS support from the host. Only the monitor component can communicate with the guest software inside the VM, and only the VMX process can make calls to the host OS. In order to pass data from one machine to another, we must connect the SP Hardware Emulator Module to the SUT’s monitor component, and then create channels to the SUT’s VMX process, to the TS’s VMX process, to the TS’s monitor component, to the TS Controller kernel module, and finally to the TS.
Controller user-mode application. The Event & Attack Module provides a hypercall interface (summarized in Table 5.4 and shown in detail in Appendix B) to the kernel components and applications in both the TS and SUT to access state and to configure the framework. This works in conjunction with the hypercall interface provided by the SP Hardware Emulation Module, which emulates new SP hardware instructions. This TF hypercall interface is also used by the TS Controller to implement its TS Controller API (Table 5.2) used by the attack scripts.

Internally to the VMM, we connect the monitor component and VMX process using a “channel” mechanism provided by the VMware code. The two sides follow a protocol we developed for the VMX–VMM channel that uses new commands for each channel call and passes data in a shared memory region. To connect the two VMs to each other through their VMX processes, we create a pair of channels, TS → SUT and SUT → TS, using new shared memory regions created via the host OS. Again we use a new protocol consisting of a set of commands and shared data to be used on these VMX–VMX channels. Each shared memory channel is unidirectional, with each machine’s VMX process occasionally polling its receiving channel to detect commands and data passed from the other machine, responding over the other channel.

Due to the use of polling, the channels currently operate asynchronously; once the other VMX process responds over the channel that a response or event is ready, a virtual hardware interrupt is sent to the guest OS (to the Controller or Proxy kernel module). Performance could be optimized by having the VMX processes signal one another within the host OS to ensure a message is consumed as quickly as possible.
after it has been generated. Currently, a VMX process must wait for the host OS to schedule the other VMX process on its own, which will then detect the message and process it. When synchronous behavior is needed by the framework, such as signaling the TS controller of a hardware event in the SUT, the entire SUT VM is frozen in its VMX process, such that the guest OS and software stack do not execute, but commands are still processed from the TS. Thus the TS can continue to issue commands that access and modify the SUT’s state through the hardware channel with no instructions being executed in the SUT that might affect that state. Only when the attack script is done responding to the particular SUT event does it send a command to the SUT VMX to resume execution (with the $\text{CONT()}$ TS Controller API call).

Virtualization/Emulation Platforms

Our framework, by using existing virtualization technology, enables reasonable performance while allowing our SUT to provide a realistic software stack and emulate new hardware. Other virtualization environments, like Xen [14], can also be used. Other simulation and emulation environments available, such as Bochs [120] and QEMU [18], could be used in place of virtualization to implement our framework as designed and described in this chapter. Each platform has certain advantages and disadvantages in terms of ease of emulation of SP features, granularity of observability and controllability, and performance. We choose a virtualization environment for performance reasons, because only parts of the hardware and protected software need to be emulated, while the OS and other non-protected software can run virtualized. VMware provided an excellent development environment, under the VMware Academic Program (VMAP).

5.4 SP Architecture and Emulation

We use the SP architecture to demonstrate the application of our framework. SP skips layers in the conventional trust chain by using hardware to directly protect an application without trusting the underlying operating system. SP protects the confidentiality and integrity of cryptographic keys in its secure persistent storage which in turn protect sensitive user data. These security properties provided by SP need to be validated. Furthermore, it is important to write and test many secure software applications for SP in a realistic environment, where the untrusted OS can be a source of attacks.

Our testing framework emulates SP’s hardware features using modifications to the VMM. While SP hardware primitives have already undergone a detailed security analysis, the framework can test the robustness of the design and its implementation, as well as discover any potential flaws. Additionally, we modify SP’s secure memory mechanisms and then show how our framework can be used to demonstrate that these new hardware features are also resilient to attack.
Table 5.4: TF Event and Attack API for Hardware Components. The full details and the hypercall interface are provided in Table B.1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Events(type)</td>
<td>Register the SUT with the Events &amp; Attack Module to have SP hardware events (SP instructions and faults) forwarded to the TS Proxy and/or the TS Controller.</td>
</tr>
<tr>
<td>Forward Events(IRQ_num)</td>
<td>Register the TS with the Events &amp; Attacks Module to receive notification of forwarded events and data messages via the specified hardware IRQ line.</td>
</tr>
<tr>
<td>event ← Get_Event()</td>
<td>Retrieves the last SP hardware event that triggered an IRQ in the TS.</td>
</tr>
<tr>
<td>Request_Read(type, *addr, size)</td>
<td>Send a request from the TS to read data (up to one page in size) from the SUT, accessing physical memory, virtual memory, general registers, or SP registers. The result is returned to the TS as a data message.</td>
</tr>
<tr>
<td>Get_Data(*data)</td>
<td>Retrieves the last data message that triggered an IRQ in the TS. Used to retrieve data that is being sent from the SUT in response to a Request_Read() call.</td>
</tr>
<tr>
<td>Request_Read_Bulk(type, *addr, size)</td>
<td>Send a request from the TS to read large amounts of data from the SUT, accessing physical memory or virtual memory only. Data is stored in a buffer file on the host.</td>
</tr>
<tr>
<td>Read_Bulk_Data(*buf, offset, size)</td>
<td>Reads data from the bulk data buffer file on the host into the buffer, buf, on the TS. Reading data from this file is much faster than requesting one page of data at a time with Request_Read().</td>
</tr>
<tr>
<td>Request_Write(*buf, type, *addr, size)</td>
<td>Send a request from the TS to write data to the SUT, accessing general or SP registers, or at most one page of memory. The data to write is given in TS memory at buf.</td>
</tr>
<tr>
<td>Send_Command(command, param1, param2)</td>
<td>Send commands from the TS to control the SUT to freeze or resume the SUT, to raise interrupts, or to set a breakpoint.</td>
</tr>
<tr>
<td>SPRegs_Get(*regs)</td>
<td>Reads or writes the SP registers to/from a data structure (regs) in the TS Proxy on the SUT.</td>
</tr>
<tr>
<td>SPRegs_Set(*regs)</td>
<td></td>
</tr>
</tbody>
</table>
The framework also serves as a platform to develop trusted software as part of the SP architecture. Any such software must be tested to be sure that it correctly uses the SP hardware mechanisms to protect itself and its data. It must correctly enforce policies and prevent leaking keys and data under physical attacks, attacks by the operating system, and interactions with untrusted software.

### 5.4.1 Secret Protection (SP) Architecture

For the testing framework, we consider the authority-mode SP architecture, as described in Chapter 3. In SP, the hardware primarily protects a Trusted Software Module (TSM), which protects the sensitive or confidential data of an application. Hence, a TSM plus hardware SP mechanisms form a minimalist trust chain for the application, bypassing a potentially compromised operating system.

SP’s hardware mechanisms consist of the following components that are relevant to the testing framework: **Roots of Trust:** SP maintains its state using new processor registers. As shown in Figure 5.5, SP uses two on-chip roots of trust: the Device Root Key and the Storage Root Hash. **Code Integrity:** The Device Root Key is used to sign a MAC (a keyed cryptographic hash) of each block of TSM code on disk. When a TSM is executing, the processor enters its protected mode — Concealed Execution Mode (CEM). As the code is dynamically loaded into the processor’s L2 cache in the protected mode, the processor hardware verifies the MAC. **Data Protection:** For the TSM’s intermediate data, while in protected mode, the TSM can designate certain memory accesses as “secure”, which will cause the data to be encrypted and hashed before being evicted from on-chip caches to main memory. This secure data is verified and decrypted when it is loaded back into the processor chip from main memory. Within the processor chip, secure data and code are tracked with tag bits added to the on-chip caches. **Interrupt Protection:** Additionally, the SP hardware intercepts all faults and interrupts that occur while in the protected mode before the OS gets control of the processor. SP encrypts the contents of the general registers in place and keeps a hash of the registers on-chip in the interrupt registers; when the TSM is resumed, the hash is verified before decryption of the registers.

Hence, to emulate SP hardware we require the following components: new processor registers (including the protected mode and roots of trust); new instructions; hardware mechanisms for code integrity checking, secure memory, and interrupt protection; and new hardware faults that these mechanisms generate.

### 5.4.2 Emulation of the SP Architecture

A traditional VMM provides a virtual machine which is (nearly) identical to the physical machine, matching the instruction set and behavior of the real CPU. It does this by trapping or translating privileged code, while ignoring hardware effects that are transparent to software, such as cache memory. Most of the time, the VMM runs code on the physical hardware, and only emulates the components that are virtualized. In order to implement and emulate new hardware architecture features, we take advantage of the VMM’s virtualization methods. For example, the VMM maintains data
structures for the virtual CPU state, which we expand to store new security registers. In the VMM, we then emulate accesses that are made to those new registers. Other useful VMM behaviors include: interception of all hardware interrupts, dynamic binary translation of code, mapping of virtual memory translations, and virtualization of hardware devices.

To emulate the SP architecture, the Security/SP Hardware Emulator Module implements the following:\textsuperscript{5}

**Protected Mode** SP requires new registers to be added to the virtual CPU, which include those for Concealed Execution Mode [102] — new mode bits and interrupt handling registers — and the roots of trust. New SP instructions are modeled as hypercalls, where the TSM running in the SUT is able to directly invoke the emu-

\textsuperscript{5}In some cases, the abstractions we implement for the Security Hardware Emulator Module vary slightly from the actual hardware implementation of the SP architecture in Chapter 3. This is necessary when non-ISA-visible components such as cache memory are modified.
lation module without going through the guest OS. Any trapping instruction would serve this purpose. Calls are then inserted into the application’s C-code using inline assembly. Alternatively, binary translation can be used by the VMM for TSM code, which can be written with new unused opcodes to implement new instructions.

**Interrupts and SP Faults** The SP architecture changes the hardware interrupt behavior when in protected mode. Since the VMM already emulates interrupt behavior, we simply detect that an interrupt has occurred during the protected mode and emulate the effect on the CPU, which includes suspending the protected mode and encrypting and hashing the general registers. To detect returning from an interrupt, the VMM inserts a breakpoint at the current instruction pointer where the interrupt occurs, so that it is invoked to emulate the return-from-interrupt behavior of SP.

As an alternative, borrowing a concept from Overshadow [31], we can setup a small block of *shim* code that triggers the return to protected mode; we provide a pointer to this shim to the VMM when starting the TSM. Whenever an interrupt occurs, the emulated SP hardware saves the interrupt return address as before, but then replaces the return address given to the OS with this pointer. Thus, when any code returns from interrupt, the shim executes instead and can make a hypercall to have the VMM resume protected mode (after first verifying and decrypting the general registers) and jump to the saved interrupt return address for the TSM [48].

Additionally, when the emulated hardware generates a new SP fault, it first reports to the TS Controller and then translates the fault into a real x86 fault, such as a general protection fault, which is raised in the SUT causing the OS to detect the failure of the TSM.

**Secure Memory** We have changed the SP abstraction of secure memory (described in Section 5.4.3). Further, we use block sizes of virtual memory pages rather than individual cache lines, since the VMM does not intercept cache memory accesses or virtualize cache behavior. While this limits the ability to model a few low-level attacks on SP (such as the behavior of cache tags), the majority of the security properties of the hardware and all those of the software can still be tested.

**Code Integrity** The TSM’s code is signed with a keyed hash over each cache-line of code and the virtual address of that line, and is checked as each cache line is loaded into the processor chip during execution. We can model this using the VMM’s binary translator to execute the TSM code. Verified instructions are tagged as secure code fragments in the dynamic binary translator cache. When the virtual CPU is executing in protected mode, it will only execute from the translator cache and will only execute fragments that are tagged “secure”. Similarly to secure memory, SP’s signing of instructions with keyed hashes is generated and stored across larger regions of code (for emulation efficiency) and is saved in a separate file. If signatures

---

6In the current implementation as of this writing, only a few interrupts are instrumented to trigger SP interrupt protection, where inserted into TSM code for testing purposes; the return from interrupt is similarly instrumented rather than detected using a breakpoint.
are generated for each page, i.e., 4096 bytes, of code, then each code fragment might require verifying one or two signatures as it is brought into the translator (depending on whether or not it crosses a page boundary).

5.4.3 Secure Memory

The original SP architecture uses two new instructions for a TSM to access secure memory: Secure Load and Secure Store. With these, any virtual address can be accessed as secure memory, where cache lines are tagged as secure (accessible only to a TSM) and are encrypted and MACed upon eviction from cache. We introduce a new secure memory model, called Secure Areas, to replace Secure Load/Store.

There are a few drawbacks to the Secure Load/Store approach. First, while most new SP instructions can be used as inline assembly, the compiler must be modified to emit the secure memory instructions whenever accessing protected data structures or the TSM’s stack. This further requires programmers to annotate their code to indicate to the compiler which data structures and variables to protect, and which code and functions are part of a TSM. Second, while a RISC architecture need only supplement a few Load and Store instructions with their secure counterparts, a CISC architecture has many more instructions which access memory rather than general registers. Third, while SP provides confidentiality and integrity for its secure memory, replay protection is also required to prevent manipulation of the TSM’s behavior, but was not explicitly described. Rather, SP assumes a memory integrity tree [169, 56, 28] spanning the entire memory space, requiring significant overhead in on-chip storage and performance when only small amounts of memory need protection.

Our new Secure Area model addresses these concerns by allowing the TSM to define certain regions of memory that are always treated as secure when accessed by a TSM. The programmer specifies the address range to protect explicitly, covering any critical data structures, allowing the compiler to use regular memory instructions without modification. This change is especially useful for our framework since the new architectural features can be tested during design-time without modifying the existing compilation toolchain. It also no longer requires duplicating all instructions in the instruction set that touch memory, a benefit for implementing SP on a CISC architecture like the Intel x86. Finally, it confines the secure memory to a few small regions, which are more easily protected from memory replay attacks with less overhead.

This change only affects how the processor identifies secure memory accesses. The underlying protection mechanism for an access determined to be secure remains unchanged, relying on on-chip cache tagging and encryption of the data sent off-chip. Only the hash generation and verification process is affected by changing the replay protection.

Hardware Design

Table 5.5 shows the new instructions added to SP to support Secure Areas, replacing Secure Load and Secure Store. The SP hardware offers a limited number of Secure
Table 5.5: New SP Instructions for Secure Areas (only available to TSM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
</table>
| **SecureArea_Add Rs1,Rs2,** region | Rs1 = start_addr, Rs2 = size
  Initialize the specified Secure Area (for region). On-chip hashes for the region are cleared. All TSM memory accesses for addr will be treated as secure if: (start_addr) ≤ addr < (start_addr + size). The start address and size must be aligned to the secure memory block size, and the size must be less than the maximum size for a region (as determined by the number of on-chip hashes and the block size). The specified region must be a valid region that is implemented by the hardware. |
| **SecureArea_Relocate Rs1,** region | Rs1 = start_addr
  Change the starting address of the specified Secure Area region. The size remains unchanged. When TSM code in multiple process contexts share memory containing a Secure Area, each may access it at a different address in their virtual address space; this is used to relocate the region. The new start address must also be aligned to the secure memory block size. |
| **SecureArea_Remove region** | Disables and clears the specified Secure Area region. On-chip hashes for the region are cleared and secure-tagged cache entries for that region are invalidated, making any data in the region permanently inaccessible in plaintext. Encrypted data in main memory remains (permanently) in encrypted form. |
| **SecureArea_CheckAddr Rd,** region | Retrieves the parameters of the specified Secure Area region. Used to verify whether or not a region is setup for secure memory and where it is located. |
| **SecureArea_CheckSize Rd,** region |                                                                                                                                              |

SecureArea regions, which the TSM can define using these instructions. Each region specifies an address range that is always treated as secure memory when accessed by the TSM, and is encrypted when accessed by any other software or hardware devices. At least two regions should be provided so a TSM can separately protect ranges of memory for its secure data and for its stack.

On-chip storage is needed for each region to track the start and end address, and to store hashes for each block within the region (shown in Figure 5.6). The block size for hashing can range from one cache line to one virtual memory page, and is
determined by the hardware implementation. The combination of the block size and the number of on-chip hashes for the region will determine the maximum size of the region in memory that can be protected.

Figure 5.6: Authority-mode SP Architecture – processor chip, enhanced with hardware support for Secure Areas and Secure Stack.

Upon defining a new region, the corresponding on-chip hashes are cleared. As secure data is written, it is tagged as secure in cache; when it is evicted from cache the contents are encrypted and a hash is computed and stored in the on-chip storage for that block. This hash must be verified when the data is read back in from off-

---

7If blocks are larger than one cache line, then multiple cache lines of data will have to be read into cache and verified in order to read or write-back any secure memory location, causing an additional delay for L2 cache misses. The additional cache lines need not necessarily be stored in the L2 cache after hash generation/verification, which might cause conflicts with other lines (and in turn possibly recursive evictions of secure lines with different tags). The security properties of Secure Memory are not affected by the choice of block size. The implementation for the SP emulator for the framework uses a virtual memory page as the block size since the VMM easily tracks accesses to memory pages but not individual cache lines.

8While not done by the SP hardware, the TSM should also zero the backing memory before calling SecureArea_Add(), since the zero hash is defined to match a zeroed memory block. Otherwise, a read or write to the uninitialized memory from a TSM will cause a data integrity fault against the zero hash.
chip memory. Since the regions can be small relative to total memory (only tens to hundreds of kilobytes are needed for our prototype TSMs), only small amounts of on-chip storage are required. Alternatively, memory integrity tree methods [28, 56, 76, 169] can be integrated to store some hashes off-chip to permit replay protection of larger regions of secure memory.

When a region is removed from secure memory protection, the previously secure data must not be revealed to untrusted software. The on-chip hashes for the region are cleared, which ensure that even the TSM can no longer get the hardware to decrypt the data since there is no longer a matching hash value on-chip. Any secure data in main memory will remain encrypted, as it was before removal when accessed by non-TSM software. Secure data remaining as plaintext in cache for the region must be invalidated, but need not be encrypted and written back to memory. These cache lines are identified by the “secure” tag, which should also include a parameter of which region it belongs to. If the region number is not stored with the tag, then any “secure” tagged cache line in the address range for that region should be cleared. If a TSM wants to reveal data, it should first copy the data to an unprotected part of memory before clearing the region.

An effect of the change to Secure Areas is that arbitrary TSM code can no longer be certain that a single load or store instruction will access secure memory. The secure region is defined by some previous section of the TSM code which may or may not have executed. Therefore, when a section of TSM code starts execution with Begin_CEM, it needs to verify that its secure regions (still) exist and that they are located at the expected address range. This is accomplished using the SecureArea_CheckAddr and SecureArea_CheckSize instructions. Once the section of code has started, it can be sure that the regions will not change until the next End_CEM instruction is reached, since the TSM will either execute continuously or will be stopped with a fault/interrupt; this would leave the suspended CEM thread in the SP registers, preventing any other TSMs from executing that could change the definitions of the Secure Areas.

If two sections of TSM code in different process contexts share the same Secure Area (using shared memory provided by the host OS to map to the same physical memory), they might access the same secure region using different virtual addresses at different times. To accommodate this sharing, each process’s TSM can use SecureArea_Relocate to reposition the region to the correct location in its address space after each Begin_CEM (along with verification of the region’s size and previous location). Since the data in the region is shared, the on-chip hashes will continue to match. Secure-tagged data in cache is still valid for the region, but at the new virtual addresses.9

9Cache tags for secure memory should be augmented such that the “secure” bit is expanded to specify the Secure Area region number (or zero for not secure), so that cache lines for different regions cannot be used incorrectly due to attack and so that they can be correctly invalidated upon removal of the area. Physically-tagged caches should add an additional tag for the offset of the page within the region to prevent reordering of pages by malicious remapping of virtual memory. Otherwise the OS could map any virtual address in the region to any “secure”-tagged line, and have it be treated as secure; the offset allows the hardware to check that the “secure” cache line is only used in the correct virtual-address location within the region. SP’s original secure memory mechanism would have required additional virtual address tags for this purpose, checked only for
Implementation and Emulation

We have prototyped Secure Areas in the testing framework in the SP Hardware Emulator. To simplify the emulation while testing the software interface, our current implementation encrypts and decrypts the entire Secure Area when entering or leaving CEM mode.\textsuperscript{10} On Begin.CEM, we decrypt any pages in the secure area which are currently mapped in guest physical memory. On an End.CEM or interrupt we re-encrypt these pages (and ensure that all pages in the Secure Area are encrypted), before returning control to the guest OS. Upon return from interrupt, we again decrypt the mapped-in pages. Since the OS cannot map or unmap any pages without taking control via interrupt, we can be sure that we will be able to re-encrypt any page that we decrypted, and that the TSM cannot access any page of secure memory that was not mapped-in when we decrypted (otherwise it will trigger a page fault, upon which we re-encrypt everything and decrypt again before returning to the TSM, but this time including the new page). For this to work, every interrupt that occurs while in CEM mode, and every return from interrupt into CEM mode, must be intercepted by the VMM to trigger SP interrupt protection. Thus the secure areas are encrypted during an interrupt in the same way that SP encrypts and restores the general registers for CEM protection.

Alternatively, we could try to emulate SP’s secure memory more closely, so pages are only decrypted as they are used. Upon entering CEM, we could unmap all of the Secure Area’s pages from the VMM’s shadow page table. Any time one is accessed, we emulate the effect of it entering the cache by decrypting the page in place, putting it back into the shadow page table, and keeping track of it in the VMM. Whenever we exit or suspend CEM, we must either re-encrypt all of these pages, or remove them from the shadow page table again, so that we may re-encrypt them if accessed outside of CEM. If not accessed again until CEM is resumed, we simply put back the mapping without ever re-encrypting and re-decrypting — much like how a real cache would behave.

In the cases of shared memory across TSMs, of multiple virtual addresses mapping to the same physical memory location, or of emulated physical attacks on memory, we must be careful that pages remain encrypted in physical memory when they might be accessed by non-TSM software. One option is that rather than mapping decrypted pages into the guest physical memory space, we might keep the encrypted versions there, but decrypt the pages to a page-cache maintained by the VMM for the Secure Area region, and only available in the shadow page table when in CEM. Thus, like a hardware cache, the virtual page would be re-directed away from the guest physical memory, and would have to be written-back eventually in encrypted form.

\textsuperscript{10}Keeping Secure Areas decrypted has the effect that our prototype does not defend against emulated physical attacks on memory while a TSM is running in CEM. This must be taken into account when modeling attacks in the framework, but would not affect a real hardware implementation of Secure Areas.
SecureArea Relocate would cause a different virtual address range to map to this same region of the page-cache.

Another alternative would be to use the technique from Overshadow [31], to maintain a separate shadow page table in the VMM for the TSM from that used by the OS and other software. When decrypted, the page would be mapped in the TSM’s copy, and when encrypted it would be maintained in the OS’s copy. Page faults cause the page to be encrypted or decrypted and moved to the other shadow page table. Switching between shadow page tables for execution would be controlled by entering and leaving/suspending CEM, without the need for the shim used in Overshadow. Again, care must be taken such that TSMs sharing the same Secure Area continue to share the latest copy of any decrypted pages and that access to the decrypted physical page from the page-cache is never available to non-TSM software (as is more easily done by real hardware with tag bits on the physical cache).

**Testing**

We have used our framework to test and debug these new interfaces for Secure Areas, to implement TSMs that use them, and to test those TSMs to verify that data is not leaked from secure data structures or from the stack (which we protect without modifying the compiler).

The framework helped significantly in the debugging process, in particular for relocating the TSM’s stack to a Secure Area. Even a very simple TSM, which only generates a derived key and saves that key in a secured data structure, manages to leak the key via the stack when using wrapper functions to access new SP instructions. The framework lets us interrupt after critical hardware operations to detect data leaked in plaintext in memory. When we find errors in the way our implementation reassigns stack pointers to use a Secure Area, we can correct the TSM code and the Secure Area setup accordingly, to ensure that all stack operations in a TSM access a valid Secure Area.

Using the framework, we also found a complication when relocating the stack to a Secure Area on an x86 platform. When an interrupt occurs in x86, the processor hardware pushes an exception frame onto the stack, using the stack pointer register; the operating system reads this frame to handle the interrupt. If an interrupt occurs while in CEM, with the stack pointer relocated to a Secure Area, the frame data will be written in the Secure Area region. If still in CEM at the time, this data will be protected as secure memory where the OS will not be able to read it. If CEM has already been suspended before the frame is written, the data will be written in plaintext and will overwrite part of the encrypted and hashed Secure Area data; when CEM is later resumed, the hash check of this region will fail. Therefore, we have developed a new mechanism to make the SP hardware aware of the stack swapping. The hardware saves the original stack pointer in an on-chip register when the stack is relocated (shown in Figure 5.6, using the new Set_Secure_StackPtr and Restore_StackPtr instructions given in Table 7.4). It will automatically restore this original stack pointer before the exception frame is written, saving the secure stack pointer on-chip; the secure stack pointer is then swapped back when CEM is resumed [48].
5.4.4 Other Architectures

While we focused on testing the hardware and software mechanisms of the SP architecture, the testing framework is by no means limited to this architecture. Other hardware security architectures such as XOM [109], AEGIS [171] and Arc3D [71] modify hardware in similar ways but have somewhat different goals and assumptions from SP. However they combine hardware and software in ways that also make them suitable for validation in our framework. Similarly, TPM [174] adds hardware to protect all software layers and provide cryptographic services. Rather than utilizing changes to the processor itself, TPM adds a separate hardware chip that integrates with the system board. This is still compatible with our testing framework, simply requiring a different set of modifications to the VMM to implement a virtual TPM device and new hooks into the Event & Attack Module. Furthermore, software-only security architectures can benefit from analysis under attack in our framework, both during development and for security evaluation. Access to existing hardware state provides insight into attack impacts and possible flaws, and provides an additional vector for injecting attacks.

5.4.5 Remote Key-management TSM

As an example of testing a TSM, we use the prototype TSM that we developed for the remote key-management scenario for authority-mode SP. We use this TSM to test the hardware features in the virtual processor (which are now protecting a real TSM). We then subject our TSM to a suite of attacks to see if the desired security properties are preserved. We attack the robustness of the TSM’s memory usage, persistent storage, network protocols, and software interfaces. We thereby use our testing framework as a testbed for testing both the software TSM as well as the hardware SP features.

For remote key-management, as in Chapter 3, we consider a trusted authority which owns multiple SP devices and wants to distribute sensitive data to them. The authority installs its remote key-management TSM on each device as well as the protected sensitive data, consisting of secrets and the cryptographic keys that protect these secrets. It also stores policies for each key, which dictate how it may be used by the local user. During operation, the TSM will accept signed and encrypted messages from the authority to manage its stored keys, policies, and data. It provides an interface to the application through which the local user can request access to data according to the policies attached to the keys. The TSM must authenticate the user, check the policy, and then decrypt and display the data as necessary. The attacker will attempt to cause the TSM to violate its access control policies or cause it to leak secrets during its operation.

5.5 Testing of SP

Testing a security architecture requires testing all components that are involved in establishing the trust chain. For the SP architecture, we use the framework to demonstrate testing the SP hardware mechanisms, the proper usage of these mechanisms by
a TSM, and the proper enforcement of access control for the remote key-management TSM. In this section we elaborate on how we use the Testing Framework to achieve these goals, testing the system’s security properties while it is under attack. In addition, we validate that the emulation of the SP mechanisms is correct and secure according to the design, as it forms the basis for the other tests.

Table 5.6 lists various attacks on the system’s security properties. The first section of these attacks address the protection of SP’s roots of trust. Data confidentiality is the primary purpose of the SP architecture. The attack generally checks to see if any sensitive data that should be protected by a TSM is ever leaked. We eavesdrop on the memory and check whether any known keys generated by the TSM, in addition to the Device Root Key (DRK) and any DRK-derived keys, are found in plaintext. This is similar to the cold boot attack [75], which looks for sensitive keys left in physical memory. If the TSM properly uses secure memory for its intermediate data, and protects its persistent data, then no keys should ever leak. Any key material in secure memory will be encrypted. We then test SP’s binding of DRK-protected secrets to the TSM — a crucial component of remote trust. The authority must be confident that any secret data it provides can only be used on the designated device and only using its TSM. This category includes any attack that tries to generate DRK-derived keys outside of the TSM or tries to replace the TSM but still access the original DRK, derived keys, or protected storage.

Table 5.6: Example Attacks on the SP Architecture Using the Testing Framework

<table>
<thead>
<tr>
<th>Type</th>
<th>Security Property</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roots of Trust</td>
<td>Data Confidentiality</td>
<td>Scan physical memory for leaks of Device Root Key, DRK-derived keys, and TSM’s other sensitive information.</td>
</tr>
<tr>
<td></td>
<td>Remote Trust</td>
<td>Attack the binding between the TSM and the DRK-protected storage. Attempt to access the secure persistent storage protected by a legitimate TSM after changing the DRK and installing a new TSM.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 5.6: (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Security Property</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP Hardware Mechanisms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access Control</td>
<td></td>
<td>Test that access to TSM secrets is available only in Concealed Execution Mode. Attempt to access roots of trust, derived keys, and secure memory in unsigned code.</td>
</tr>
<tr>
<td>Securing General Registers on Interrupts</td>
<td></td>
<td>Attack the general registers during an interrupt of a TSM through eavesdropping, spoofing, splicing, and replay attacks.</td>
</tr>
<tr>
<td>Code Integrity</td>
<td></td>
<td>Attack TSM code during execution through spoofing and splicing; attack TSM code on disk.</td>
</tr>
<tr>
<td>Secure Memory</td>
<td></td>
<td>Attack intermediate data of TSM through eavesdropping, spoofing, splicing and replay; attack the use of secure memory for TSM’s data structures or stack.</td>
</tr>
<tr>
<td>Secure Storage</td>
<td></td>
<td>Attack the TSM’s use of the SRH to protect the integrity of its secure storage for persistent data (splicing, spoofing &amp; replay attacks).</td>
</tr>
<tr>
<td>Generic TSM Attacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control Flow Integrity</td>
<td></td>
<td>Attack TSM’s indirect jump targets that are derived from unprotected memory. Arbitrarily modify jump targets within the TSM (to the entry points of other basic blocks).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Attack the input data for branch conditions in the TSM from unprotected memory. Replay secure data to cause incorrect branch decisions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Attack TSM entry points by entering CEM at arbitrary points in the code, skipping access control checks or initialization of secure memory.</td>
</tr>
<tr>
<td>TSM Page Mappings</td>
<td></td>
<td>Remap TSM code pages and data pages, as a means to attack secure memory or control flow.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 5.6: (continued)

<table>
<thead>
<tr>
<th>Type Property</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM Specific Attacks</td>
<td></td>
</tr>
<tr>
<td>Key-chain management</td>
<td>Attack communications from the authority or the communication protocols. Spoof key add/delete message; replay key-add message after it is deleted; corrupt a key-management message in transit.</td>
</tr>
<tr>
<td>Access control on keys</td>
<td>Attack the TSM’s policy enforcement. Exceed usage limits/expiration of keys; attempt to use a key that was deleted; attempt to perform a disallowed operation with a key.</td>
</tr>
</tbody>
</table>

The second section sets up a series of attacks on the basic mechanisms of SP, such as access control on SP instructions, code integrity checking, and encryption of secure data in protected mode. These tests verify that the emulation is correct and also validate the original security analysis. For example, we attack SP’s Concealed Execution Mode by attempting to modify registers during an interrupt. A non-TSM application’s registers can be modified by a corrupted OS without detection, causing changes in the application’s behavior. However, a TSM will have its registers encrypted and hashed by the SP hardware upon any interrupt, such that SP detects the modification when resuming the TSM. Similarly, a non-TSM application’s registers can be eavesdropped by a corrupted OS, but since SP encrypts the registers before passing control to the OS, this breach of confidentiality is prevented.

We also test how TSMs use SP mechanisms to protect intermediate and persistent data. We test our new secure memory implementation to verify that a TSM compiled with GCC can adequately protect its intermediate data on the stack. Our example TSM in Figure 5.7 derives a new key from the Device Root Key and uses it for encryption. The attack script sets up a breakpoint to detect that the Encrypt() function is about to use the key. The script then freezes the SUT and scans physical memory to look for the derived key. When we first implemented this attack, the script found that the key had been leaked via parameter passing on the stack, violating data confidentiality. As a result, we have implemented a new software mechanism, used within a TSM just after Begin_CEM, to swap the TSM’s stack to use memory in a designated Secure Area. The same attack script then verifies that this modified TSM correctly protects the confidentiality of the key when passed as a parameter. This demonstrates how a secure hardware mechanism (e.g., for secure memory) can be used incorrectly by a TSM, often inadvertently, leading to vulnerabilities. In Appendix A, we offer a list of best practices to improve security when developing TSMs.

The third section in Table 5.6 shows generic attacks on a TSM, which test security properties common to most TSMs (e.g., control flow, entry points). These attacks
consider that a basic goal of many TSMs (and indeed of the SP architecture) is to provide confidentiality and integrity to any sensitive information and enforce access control.

We develop tests of the robustness of the TSM against future unknown vulnerabilities that might arise in the hardware or TSM code. Since the penetration mechanism is unknown, we instead model the effects of the attack. For example, the control flow of the TSM could be attacked in many different ways. When the TSM makes branching decisions, the jump targets and the input data for the branch conditions should be protected. If either is not stored in secure memory, or if secure data can be modified or replayed, then arbitrary changes to the TSM’s control flow would be possible. We verify that a TSM only bases control flow decisions on data in its secure memory, and test how control flow violations could cause data to leak.

As another example, we consider control flow attacks that allow arbitrary entry points into a TSM. Since instructions to enter protected mode (Begin_CEM) are not signed, Begin_CEM could be injected into the TSM to create an entry point. We implement this as an attack script, crafting a case where the TS overwrites instructions and tries to enter in the middle of a TSM function without detection, bypassing access control checks. This attack was not addressed in the original SP architecture[102]. To prevent the attack, we add a new security requirement to SP that it must distinguish entry points in TSM code from blocks of code that are not entry points. This can be achieved by adding an extra bit to the calculation of the signature of each block of TSM code, indicating whether or not it is an entry point.

The attack on TSM page mappings demonstrates a system-level attack, where rather than attack the TSM directly, the OS manipulates the system behavior to indirectly affect how the TSM executes. The OS can manipulate process scheduling, intercept all I/O operations, and in this case, modify how virtual addresses map to physical addresses. Dependencies in the TSM code or improper input validation could cause vulnerabilities that the OS can exploit in this manner.

In general, generic attacks can be developed by considering desired invariants for the TSM’s information flow, program behavior, or security properties. Then the framework can be used to test for these conditions while running various test cases, based on known inputs, random inputs, or carefully crafted inputs (e.g., fuzz testing [68]). The framework will then help identify security vulnerabilities and design flaws during the process, as we have demonstrated. Formal verification techniques discussed in Section 5.6 or other mathematical models of program behavior or security properties could be used to help determine the tests and the program inputs.

The last section in Table 5.6 shows application-specific attacks for a particular TSM — in this case our Remote Key-management TSM (which makes up part of the Emergency Management TSM in Section 6.6.3). This TSM stores cryptographic keys, security policies, and secure data in its persistent secure storage, which it protects using SP’s underlying hardware mechanisms. We test the confidentiality and integrity

---

11In some cases, this attack would be detected by SP — if the injected instruction is not correctly aligned to the start of a block of signed code. It might also be detected if later in execution the TSM jumps back to code before the injection site. When execution then reaches the injected code, the code integrity checking would raise an exception for the unsigned Begin_CEM instruction.
of the storage itself, the TSM’s use of the storage to protect keys and key-chains, and its enforcement of the policies on accesses to data that the keys protect. We also test the protocols the TSM uses to communicate with a remote authority, managing the keychains.

Our system implements the SP hardware mechanisms, a full TSM providing an API to the testing application (see Table 6.1), and a suite of attacks that test both the software and hardware components using our new testing framework. This is a major step toward the complete validation of the design of the SP architecture together with its applications. Furthermore, we demonstrate that TSMs must be carefully written to avoid serious security flaws, and that a security architecture can benefit from testing with many different applications. Our framework provides a platform for this necessary testing.

<table>
<thead>
<tr>
<th>Application with TSM (TSMapp)</th>
<th>Attack Script (pseudocode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_CEM</td>
<td>EXECUTE(TSMapp, params)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>nonce ← Hash(C_ENC, KeyID)</td>
<td>// Wait for key generation</td>
</tr>
<tr>
<td>Reg1 ← DRK_DeriveKey(nonce)</td>
<td>EVENTADD(DRK_DeriveKey)</td>
</tr>
<tr>
<td>SecureMem.AESkey ← Reg1</td>
<td>EVT ← WAIT()</td>
</tr>
<tr>
<td></td>
<td>// Read the generated key</td>
</tr>
<tr>
<td></td>
<td>ACCESS_SPREG(r, SPRegs)</td>
</tr>
<tr>
<td></td>
<td>SPKey ← SPRegs.CEMBuffer</td>
</tr>
<tr>
<td>// Attack script injects a breakpoint at start of Encrypt function</td>
<td></td>
</tr>
<tr>
<td>Ciphertext ← Encrypt(SecureMem.AESkey, &amp;SecureMem.data, sz)</td>
<td>// Inject breakpoint for Encrypt()</td>
</tr>
<tr>
<td>END_CEM</td>
<td>BREAKPOINT(“&amp;Encrypt”); CONT()</td>
</tr>
<tr>
<td></td>
<td>// Wait for interrupt due to breakpoint</td>
</tr>
<tr>
<td></td>
<td>EVT ← WAITFOR(Interrupt)</td>
</tr>
<tr>
<td>// Send encrypted file on network or store on disk</td>
<td></td>
</tr>
<tr>
<td>Network_Send(TTP, Ciphertext, sz)</td>
<td>// Scan phys. memory for leaked key</td>
</tr>
<tr>
<td>...</td>
<td>for addr = 0 to 256M do</td>
</tr>
<tr>
<td></td>
<td>ACCESS_MEM(PHYS, r, addr, 4096, buf)</td>
</tr>
<tr>
<td></td>
<td>if strstr (buf, SPKey) then</td>
</tr>
<tr>
<td></td>
<td>return “Derived Key Leaked in Memory”</td>
</tr>
<tr>
<td></td>
<td>return “Derived Key Not Found in Memory”</td>
</tr>
</tbody>
</table>

Figure 5.7: Example Application and Attack Script for Detecting Leaked Keys

---

12 We have implemented most of the attacks in Table 5.6 except for the generic TSM attacks and the code integrity attacks.
5.5.1 Testing Example

Figure 5.7 shows a sample TSM on the left, and a corresponding attack script using the TS Controller API (Table 5.2). This demonstrates the interactions between the TS and SUT for event detection and modification of SUT state. As described previously, the simple attack shown here verifies that secure data (here the derived AES key), placed on the stack by the TSM as a function parameter, is not leaked in physical memory where the OS could read it. This attack demonstrates precise coordination of software events (injected breakpoints) with access to the hardware (physical memory state), while the SUT is frozen to prevent clearing or overwriting of any data in memory. The script also requires access to the internal state of the SP hardware from the TS to verify the results of the attack.

Based on our experience so far, we expect that most attack scripts will be longer and will involve many additional steps and interactions, along with a complete TSM and its corresponding application. The full range of events and attack mechanisms described in Table 5.1 are available to the attack scripts, with the TS in full control over the applications, OS, and hardware running in the SUT.

5.6 Related Work

Our framework is designed for testing the security properties of security architectures and for the development of trusted software for such enhanced architectures. Since our testing framework emulates new security architectures and runs full system software, it can be compared against hardware simulators and full system simulators.

Micro-architectural simulators like Simplescalar [13] are cycle-accurate and hence can be very useful in estimating performance metrics, but they cannot simulate a realistic software system with a full commodity OS. Thus it is impossible to test the security critical interactions of a software-hardware security solution with such a simulation architecture.

Chow et al. [34] use system emulation to passively trace data leaks in applications. However, our framework also performs active attacks and looks for violations of security properties. Chow’s work also does not consider violations other than data leaks, while we consider more security properties, such as data integrity, policy enforcement, and control flow. Furthermore, we are looking for flaws in trusted code and hardware mechanisms that are specifically designed to protect security, unlike Chow where the applications are tested for properties they were not designed for, therefore leading to unexpected results.

Virtual machine introspection [66, 136, 111] techniques, described previously, provide access to VM-state in similar ways to our framework, however, they focus mostly on observability of software configurations or low-level operating system and hardware behavior. Examples include intrusion detection and virus-scanning from non-vulnerable host systems, preventing execution of malware, and tracing memory or disk accesses. Instead, we strive to combine observability of the full-system state with controllability of those same components, actively during operation, to attack
software thought to be secure. In the related work, the focus is on techniques for security monitoring of production machines, rather than design-time testing of new architectures or of new software systems to evaluate their potential vulnerabilities and flaws. Where some of these techniques provide improved hooks into the virtual machine monitor [137], the hooks could be integrated into our framework to make our attack scripts more robust and more flexible.

The efforts by IBM [19], Intel [154] and others [167] provide the functionality of a virtual TPM device to software, even when the physical device is not present. In contrast, we not only emulate the new hardware but also hook into the virtual device to observe and control its behavior for testing purposes, and study the interaction with other hardware and software components.

Another related area of research is the formal verification of both hardware and software, in which formal mathematics is used to write specifications for computer hardware and software, and proof techniques are used to determine the validity of such specifications. The complexity of formal verification problems range from NP-hard to undecidable [85, 143, 91, 82]. The complexity of these formal verification mechanisms led to the use of hybrid techniques [21] which use some formal as well as informal methods. Some formal methods of verification include theorem provers (e.g., ACL2 [124], Isabelle/HOL [129]), model checkers [113], and satisfiability solvers [179, 42]. Some informal techniques used in practice are control circuit exploration, directed functional test generation [63], automatic test program generation [36], fuzz testing [68], and heuristic-based traversal [26]. The formal and hybrid techniques try to verify the hardware and software separately, unlike our holistic verification of a software-hardware system.

The limitation of the formal verification techniques is that they must verify each component piece by piece. This is necessary since the complexity of both specification and verification explodes exponentially with addition of more pieces to be tested. In our approach, we verify the system in an informal but systematic and efficient way, and consequently we can model both the security critical hardware and software together; we are thus better able to determine the security impacts of the interactions of the various components.

5.7 Summary

We have designed and implemented a virtualization-based framework for validation of new security architectures. This framework can realistically model and test a new system during the design phase, and draw useful conclusions about the operation of the new architecture and its software interactions. It also enables testing of various software applications using new security primitives in the hardware or in the OS kernel.

Our framework serves as a rapid functional prototyping vehicle for black-box or white-box testing of security properties. It can utilize and integrate multiple event sources and attack mechanisms from the hardware and software layers of the system under test. These mechanisms can test both low-level components and high-level
application behavior. As a result, a comprehensive set of attacks are realizable on
the hardware, operating system, and applications.

We implement the SP architecture in our framework and test its security mecha-
nisms thoroughly, studying the interactions of trusted software with the hardware
protection mechanisms. We also improve the design of SP’s secure memory for both
emulation and a hardware implementation. We implement a TSM for remote key-
management to demonstrate a real application of the SP architecture. Using a suite
of attacks on each layer of the architecture, we thoroughly test each component of
SP’s trust chain.

Future work includes testing more applications and their partitioning into Trusted
Software Modules in SP (e.g., a TSM that enforces mandatory and discretionary
access control policies on documents developed by other members of our research
group [33]), random attack generation, and devising additional cross-layer system
attacks. We hope to have provided a framework to facilitate more systematic design-
time testing and reasoning about security properties.
Chapter 6

SecureCore Platform

6.1 Introduction

In previous chapters, we have focused on how SP can be used to protect keys and critical secrets against attacks on confidentiality and integrity. We use new secure hardware features and trusted software to ensure these secrets are never revealed, even under the assumption that the operating system cannot be trusted. In this chapter, we change this assumption to examine how to integrate SP with a trustworthy operating system. We approach this to achieve defense-in-depth, where the operating system protects information flow and the dissemination of plaintext data that has been provided to the local user, but SP still protects the use of the keys themselves in the event that the operating system fails. We also take advantage of the operating system to provide trusted I/O with the user for high integrity display of data and for user authentication—both of which enable more interesting and more secure uses of TSMs (Trusted Software Modules).

Generally speaking, the information flow we want to protect is the containment and revocation of plaintext data provided by a TSM. Without a trustworthy operating system or dedicated trusted I/O hardware, a TSM is limited since it must go through the operating system and the untrusted part of the application to interact with the user. Thus the operating system and application have an opportunity to intercept or modify any output data or any input to the TSM, and can freely save or disseminate the information. As part of the SecureCore project [101], we have integrated SP with a trusted software stack, including a trusted separation kernel, operating system, and I/O application. This SecureCore software stack [90] enforces Mandatory Access Control (MAC) policies, such that information flows that might allow dissemination of the TSM’s data can be controlled, and provides a trusted path to the user of the device.

Using the SecureCore platform [104, 106], we expand our motivating example from Chapter 3 to be a more comprehensive, handheld Emergency-device (an E-Device), which provides both emergency and non-emergency data to first responders at multiple security levels, while ensuring the containment and integrity of data and providing the ability to use familiar commodity software.
In addition to an emergency device for crisis response use, this device is also designed for possible military use. In this case, the emergency situation might be a military operation with strict confidentiality requirements and strict separation requirements for Mandatory Access Control. The central authority will be a secure military facility, hosted in multiple locations to avoid loss of communications under an attack. Operational plans and other data may need to be withheld until the operation actually begins. Given this scenario, containment of data and control over the emergency state of the device is of particular importance in motivating the design decisions for SecureCore.

Previously, the authority could revoke access to keys and instruct its TSM to control access to new data. In the expanded E-Device, the authority can additionally revoke future access to data that was already displayed, saved in plaintext, or put to use by application software within the device. Thus, once an emergency event is over and prior data that was legitimately accessed is no longer necessary, all sensitive data that was once shown to the user can be removed and contained within the E-Device.

For containment, the trusted separation kernel provides multiple execution environments, called partitions, each at a different security level. Partitions are like virtual machines, some of which can run commodity operating systems and software, while others can run trusted operating systems. The kernel maintains strict separation between the partitions, allowing information flow only according to MAC policy. To do this, it virtualizes hardware components and provides controlled services to each partition. These services, such as networking and disk access, will be limited or isolated to ensure that access policies cannot be violated. The trusted partition can provide secure I/O for the user, while regular partitions can be created to contain emergency and non-emergency data. We take advantage of both the containment and the trusted path to the user.

We integrate SP’s trusted software modules into a number of the components in the SecureCore software stack to enable the authority to control and configure the device, send data and policies securely to the correct partitions, and to manage emergencies. These new TSMs work in conjunction with traditional authority-mode TSMs, which provide controlled access to emergency data to the user, but now with improved revocation capabilities, additional communication channels, and better control over the use of information after it leaves a TSM. We also use TSMs to improve the security of the SecureCore platform itself utilizing SP’s hardware roots of trust in the security kernel, adding resilience to physical attacks. We provide improved security for authority communications to both emergency and non-emergency partitions and protect the kernel and its persistent data from physical attacks that could violate MAC policy.

To enable this integration, we have also made improvements and changes to the SP architecture. We provide a mechanism by which TSMs can be used in both user space and kernel space, making calls between processor protection domains without invoking untrusted software. We also discuss the need to virtualize the SP architecture, which is then solved in Chapter 7.

Finally, to conclude this work, we have implemented a prototype emergency-response TSM that performs remote key-management services and demonstrates
emergency state management features to be used in the E-Device. This prototype has been integrated into the prototype SecureCore software stack and has been tested against simulated attacks on the local device and on communication with the authority. It additionally serves as a proof-of-concept TSM that provides useful security services to an application and to the authority via its API and network interface.

Parts of this chapter are taken from [104].

6.2 Background on SecureCore Software Stack

The existing SecureCore software stack [90, 106] builds upon the layered privilege levels of a typical operating system, adding additional higher-privileged layers beneath the operating system for a least-privilege separation kernel and a trusted services layer, together referred to as the Trusted Management Layer (TML). A commercial processor, like an x86, is assumed, with the ability to provide the additional privilege levels in hardware with its virtualization support [126, 88, 3]. The architecture of the SecureCore software stack is shown in Figure 6.1.

![Figure 6.1: Baseline SecureCore Software Architecture [104]](image-url)

1The SecureCore project was joint work between Princeton University, the Naval Postgraduate School, and the USC Information Sciences Institute (the authors of [104]).
The TML strictly divides the processor’s resources to provide an isolated environment for each of the partitions. For example, time slices for CPU usage are pre-configured and strictly enforced to avoid covert timing channels between the partitions; other resources like disk and memory are appropriately subdivided and virtualized to avoid overt and covert communication between partitions, except where explicitly permitted and setup by the TML.\textsuperscript{2} Hardware resources can also be dedicated to a particular partition when sharing is not necessary, with other partitions unaware of the resource or its use.

Most partitions run commodity operating systems and software, and are assigned a single security level for MAC policy, i.e., Top Secret, Secret, Confidential, or Unclassified in a multilevel security (MLS) \cite{17} policy. Additionally, one or more emergency partitions can be created, also running commodity software, to be used only for emergency access to data. A final, trusted partition runs a special trustworthy operating system, in this case called the Trusted Executive, which can be trusted to operate at multiple security levels simultaneously and can direct information flow (e.g., declassification of data) between the other partitions according to predefined MAC policies.

The trusted partition also runs the Trusted Path Application, which manages passwords, user authentication, and other secure interactions with the user. It additionally determines partition focus — the individual partition that has access to the display, keyboard, and mouse at a particular time. At any time, the user can press the \textit{secure attention key} (e.g., Ctrl-Alt-Del), which immediately switches to the trusted partition and the Trusted Path Application. From there, the user can select a new partition to have focus, authenticate to the system, or shut down the device.

As the device leaves the authority’s depot after initialization, the partitions, MAC policy, and allocation of resources are pre-configured in the TML and generally cannot be changed during operation. After deployment, the authority can remotely communicate with the device to manage the emergency state and send new data to the user (in addition to any SP-specific communications between the authority and device).

All communications with the authority are managed by the TML, through a component called the Trusted Channel Manager. The channel manager allows the authority to establish a connection to the TML on the device, which can then be extended to the partitions or other secure components (e.g., the Trusted Path Application). Thus the authority can either remotely configure and manage the device itself, communicate directly with the user via a trusted path, or send data and other messages to specific partitions at the appropriate security level.

In particular, the authority will remotely manage the emergency state of the device. By default, the E-Device is configured to be in non-emergency mode, under which the emergency partition is disabled and cannot be selected by the user for partition focus or used to access data. When an emergency event occurs, the authority will connect to the device through the Trusted Channel Manager and send

\textsuperscript{2}Covert channels \cite{99} are mechanisms that can be used to as communication channels that were not intended as such, especially when used to transmit information in violation of access control policies.
an emergency-state message to activate the partition. The user will be notified of this event and can then press the secure attention key to switch to the emergency partition. The authority can also send text-only messages to be displayed securely to the user by the Trusted Path Application. Since these messages use only trusted channels and trusted software, the user can be assured of their integrity.

After the emergency ends, the authority can again communicate with the TML and instruct it to disable the emergency partition. The data it contains can be maintained for later analysis and auditing, or the authority can direct the TML to reset the partition to its initial state from before the emergency, thus permanently erasing any data that was revealed within the partition during the emergency.

6.3 Protection of Emergency Data

We use authority-mode SP to protect emergency data on the E-Device and to secure communications with the authority. We add TSMs inside the emergency partition and inside the TML to protect the trust chain, securing execution and links from the remote authority to the E-device and to its emergency partition. These TSMs provide secure communication channels with the authority, protect emergency state management, and provide trustworthy display and mechanisms for revocation of sensitive data.

6.3.1 Threat Model for Protection of Emergency Data

The processor hardware, TML software layers, and TSM code together form the Trusted Computing Base (TCB) of the E-Device. Their operation on the E-Device alone or in communication with the authority and third parties is governed by the following threat model.

Keys and other data for emergency use on the E-Device are initially protected by TSMs and subject to TSM-enforced access control policies of the authority. Such data contained by a TSM is subject to the SP threat model discussed in Chapter 3, which includes hardware attacks and software attacks at all times.

The rest of the system is subject to a slightly more relaxed threat model, given that it controls somewhat less critical data and operations and is not supported with hardware security mechanisms. This includes data and keys that are made available to the non-TSM components of the software stack, and includes the code that makes up the software stack. We primarily consider attacks from untrusted software components and attacks over the network. Trusted software is assumed to operate correctly, but its code as well as other data on disk must be protected when the device is offline.

The authority may also send additional keys and data to the device to be used in the emergency partition. This would take place with a separate communication channel to the device.
More generally, physical hardware attacks for non-TSM components are considered only when the device is offline or powered down.\footnote{When the device is powered down, attacks on the confidentiality and integrity of storage through physical access become simpler to perform compared to online interception and modification of storage and memory accesses during operation.} All trusted system software must be integrity checked upon bootup, and all data for the TML and the partitions must be encrypted and signed with a keyed hash until a legitimate user boots up the device. Once in operation, a legitimate user is trusted to keep the physical device secure, and the trusted software layers can rely upon the mechanisms in the processor hardware to achieve separation from untrusted code. These include privilege levels, virtual memory, and similar hardware mechanisms; off-chip memory, system buses, and other hardware components cannot be probed during operation.

During operation the threat model for the TML is that applications of the TML, including guest OSs other than the trusted executive, are not trusted to conform to its policies, and may in fact be hostile. For example, at runtime, the software executing above the TML may attempt to access the keys that the TML uses to establish secure channels, and application software or the commodity operating systems may attempt to write emergency data to a location outside the emergency partition or attempt to access high integrity information through low integrity mechanisms. By enforcing MAC policy, the TML prevents all such attacks, maintaining proper isolation and information flow, and therefore confidentiality of data within each partition. The trusted executive is trusted to manage its applications in a manner that does not introduce covert channels between applications that are at different security levels.

For network traffic, we assume the standard Dolev-Yao model [46], that arbitrary parties can capture, modify or insert network traffic. Denial of service attacks on network traffic are not considered.

We assume that the E-Device is initialized securely with TML-, TSM- and SP-specific keys by the authority. We also assume that the third parties and the authority securely exchange the required keying material and protect their own keys from exposure.

TCB software at the authority, third parties and in the E-Device is assumed to behave correctly and securely. However, application TSMs are subject to inconsistencies in their execution environment, such as denial of access to the processor, as their execution is dependent on untrusted software that controls physical resources. The TML will ensure that each partition is scheduled properly, that MAC policy is enforced between partitions, that there are no covert channels introduced between partitions, and that partitions cannot disrupt the execution of the TML itself, but software within untrusted partitions is not protected from denial of service attacks.

### 6.3.2 Secure Communication Channels

Our concept for emergency response involves a network of participants, including a coordinating authority, the expected first responders, and third party data providers who maintain information that is expected to be useful during an emergency. The authority manages the distribution of keys and policies via its own secure computing
facility and coordinates emergency response for a given crisis, including alerting all entities of the emergency and disseminating emergency data to first responders.

Distribution of this emergency data must take place on secure channels over the network. We consider a channel to be a secure channel if it uses a secure protocol, the protocol is implemented correctly, and the channel endpoints are secure against both the modification of their behavior and against unauthorized disclosure of channel and keying information. A secure communications protocol protects against message content disclosure and modification, as well as traffic analysis and insertion or deletion of packets.

On the E-Device, the Trusted Channel Manager establishes, configures, and secures all communications channels to the device and acts as either an endpoint or intermediary for the channel. It stores or generates the cryptographic keys necessary to setup the channels, authenticates the remote endpoint, and determines which software component on the device is the appropriate endpoint for the channel — conforming to both MAC policy and discretionary access control policy. The Trusted Channel Manager resides inside the TML as a trusted component, and is therefore able to setup connections with all partitions at any security level. A TSM is added to safely store communications keys and access control policies on the device in the SP secure local storage and to have access to the DRK-derived secrets needed to setup secure channels with the authority.

The E-Device supports several different forms of secure communications channels, distinguished by their endpoints, which provide emergency systems designers with flexibility in constructing new systems [104]. These are depicted in Figure 6.2.

The simplest is a Trusted Channel, which is a secure channel between two TCBs (e.g., a TML or a trusted system) on different systems [24]. The Trusted Channel Manager maintains cryptographic keys to secure this channel and to authenticate the remote party. This type of channel can be used directly to send new configuration data to the TML. All other channels over the network are built on top of Trusted Channels by extending the endpoint on the local or remote device. For example, an Extended Trusted Channel is a secure channel created by extending the I/O interface of a Trusted Channel to the TML interface of a given partition, which allows applications to securely interact with a remote TCB.

A TSM-TSM Channel provides cryptographic assurance against message disclosure and modification between application TSMs, e.g., on different machines. On the E-Device, the Trusted Channel Manager sets up an Extended Trusted Channel to the emergency partition. The guest OS in that partition then extends the channel to the TSM. The TSMs at each endpoint further encrypt and hash the contents of the communication with DRK-derived keys and then send it over the Extended Trusted Channel. This protects the data from the guest OS as well as the TML which, while trusted, may be less secure than the TSMs themselves.

Channels can also be setup within the device to the local user. These channels connect various components to the trusted partition and its Trusted Path Application, which provides high integrity input and display, protected from modification such that the user can be sure any messages are legitimate. For example, a Trusted Path is a secure channel between a user and the TML on the E-Device. A Trusted Application
<table>
<thead>
<tr>
<th>Channel Type</th>
<th>Path</th>
<th>Example Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Channel</td>
<td>$C \leftrightarrow D$</td>
<td>Relay encrypted data between devices</td>
</tr>
<tr>
<td></td>
<td>$C \leftrightarrow I$</td>
<td>Send configuration or emergency data</td>
</tr>
<tr>
<td>Extended Trusted</td>
<td>$B \leftrightarrow C \leftrightarrow D$</td>
<td>Share data with ‘Emergency’ label between two devices</td>
</tr>
<tr>
<td>Channel</td>
<td>$B \leftrightarrow C \leftrightarrow I$</td>
<td>Send new encrypted emergency data to the storage in the emergency partition</td>
</tr>
<tr>
<td>TSM–TSM Trusted</td>
<td>$A \leftrightarrow B \leftrightarrow C \leftrightarrow D$</td>
<td>Exchange emergency information between two trusted users</td>
</tr>
<tr>
<td>Channel</td>
<td>$D \leftrightarrow E \leftrightarrow F$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A \leftrightarrow B \leftrightarrow C \leftrightarrow I \leftrightarrow J$</td>
<td>Distribute new policies to devices in the field</td>
</tr>
<tr>
<td>Trusted Path</td>
<td>$C \leftrightarrow U$</td>
<td>Invocation of the secure attention key</td>
</tr>
<tr>
<td>Trusted Application</td>
<td>$A \leftrightarrow B \leftrightarrow C \leftrightarrow U$</td>
<td>Display emergency messages with high integrity</td>
</tr>
<tr>
<td>Display</td>
<td>$H \leftrightarrow G \leftrightarrow C \leftrightarrow U$</td>
<td>User authentication</td>
</tr>
</tbody>
</table>

Figure 6.2: Trusted Channels in the SecureCore Architecture (developed by [104])
Display, discussed below, enables an application (i.e., the emergency partition TSM) executing in the context of an insecure OS to securely write to the screen, which is managed by the TML.

### 6.3.3 Emergency State Management

The authority maintains a binary global emergency state, i.e., ON or OFF, and notifies the authorized E-Devices of any state changes. The E-Devices may grant access to emergency information if the state is ON and must deny access when OFF.

When an emergency is declared, the authority sends state-change notifications to the E-Devices. Once the TML interprets the message, it prompts the user to access the emergency partition, via the Trusted Path Application.

While the emergency is in effect, users can access any active partition that they are cleared to see, including the preconfigured emergency partition. Within the emergency partition, finer granularity application-specific access controls on emergency data may be provided by an Emergency Management TSM in the application domain.

When the emergency is over, the authority announces a change to the global emergency state, prompting the TML to start the emergency closure process. It displays an end-of-emergency message, which prompts the user to change focus to the trusted partition (to be completed within a configurable period), revokes access to the emergency partition, and restores the emergency partition to its original state.

These changes to the emergency state are transmitted by the authority as emergency state management messages. State management consists of the following steps: (1) message generation; (2) message transmission; and (3) message processing on the E-Device. Each of these steps needs to be trustworthy to ensure consistent and correct emergency state management.

In addition to the emergency state, the authority maintains a counter of the number of state changes it has issued; also, it maintains a record of the acknowledged state and counter values for each E-Device, along with their DRK values. When the authority changes the global state, it must securely synchronize with each E-Device. On the individual E-Device, local emergency state and a state transition counter are maintained by a state-management (E-State) TSM in the TML.

When the authority declares an emergency, it increments its counter associated with the E-Device and generates an emergency state management message for each E-Device that consists of: (a) a command type indicator (indicating a state update message); (b) a payload of the new state and counter, encrypted with an encryption key derived from the DRK; (c) a signature (cryptographic keyed hash) of the encrypted payload and command, using a signing key derived from the DRK; and (d) two nonces used to derive the encryption and signing keys.

The emergency state management message is sent to the E-Device through a Trusted Channel to the device’s TML. Only the E-Device to which the emergency
state management message was intended is able to successfully process the message.\textsuperscript{5} The E-State TSM independently\textsuperscript{6} verifies the originator of the state management message. For emergency state management, two functions, \textit{Update State} and \textit{Get State} are implemented on the E-Device. The \textit{Update State} function checks the counter and the integrity of the message using the signature. It also creates a response to the authority by generating a signature (a MAC) over the message using a signing key derived from the signing nonce and the DRK. The E-State TSM sends the signature back to the authority over the trusted channel, but does not need to send the message payload or nonce, since the authority already has the initial update message.

The TML, via its TSM, uses \textit{Get State} to retrieve the new state and use it outside of the TSM to configure the partitions and other policy enforcement. To ensure that the update of emergency state is trustworthy, only the TML can pass update messages from the trusted channel with the authority into the E-State TSM, and only TSM’s executing in the TML, such as the E-State TSM, can invoke \textit{Update State} and \textit{Get State}.\textsuperscript{7}

The above functions are implemented in the E-State TSM; the current emergency state and counter are stored in the secure local storage, in the SRH register if not used for secure local storage, or in a similar non-volatile data register available only to TSMs. For high-threat deployment environments, enhanced assurance is provided by a version of SP that includes state management primitives in its instruction set architecture. Instead of implementing them in the E-State TSM software, this version includes registers for a state variable and a state transition counter, as well as instructions for the \textit{Update State} and \textit{Get State} functions. With these hardware enhancements, even a TSM does not have the ability to directly modify the emergency state on the device.\textsuperscript{8}

The E-State TSM is of course not limited to maintaining only the binary emergency state and the state transition counter; it can maintain additional more detailed emergency state in the secure local storage, such as the type of emergency, severity of the emergency, or scope of the emergency. This additional data may be used by the TML and the Emergency Management TSM to affect access control policy.

\textsuperscript{5}The TSMs could alternatively be designed to permit broadcast messages from the authority, using public-key signatures. Thus, state management messages would not need to be transmitted one-by-one by the authority to each device.

\textsuperscript{6}Decoupling the channel authentication from message authentication allows for flexibility to incorporate ad-hoc and/or peer-to-peer transmission of emergency state management messages in the future.

\textsuperscript{7}The TSMs will be written such that only the E-State TSM ever makes calls to \textit{Update State} and \textit{Get State}. Since TSM code is signed, no other TSMs will have code to call these functions. Furthermore, the functions themselves will check the privilege level of the processor and only permit use when in the appropriate mode, in this case restricted to the TML’s privilege levels.

\textsuperscript{8}In this case, where hardware is used to maintain the emergency state, the DRK should be used directly by the authority to sign the emergency state management messages and by the hardware on the E-Device to verify the messages. Otherwise an exploit could cause a TSM to use DRK-derived keys to sign spoofed state management messages and pass them to the hardware instructions without first receiving them from the authority. Non-volatile registers will be used for the state variable (1 or more bits, depending on the number of states supported) and the state transition counter (64 bits).
6.3.4 Containment of Emergency Data

The organizational security policy enforced by the E-Device requires that emergency information from the data providers only be accessible to authorized users acting within the emergency partition, and only during a proper emergency declared by the authority. MAC policy enforcement (by the TML) and discretionary policy enforcement (by the Emergency Management TSM) jointly restrict information flows on the E-Device before, during and after the emergency.

Emergency data is installed at the authority’s secure facility or sent to the E-Device from the authority and data providers over Trusted Channels, and is confined within the emergency partition. The data may be further encrypted so that it can only be accessed by the Emergency Management TSM application, which can enforce finer granularity access policies within the emergency partition.

When transmitting emergency data, the authority establishes an Extended Trusted Channel to the emergency partition. The emergency partition and the authority are allocated an “emergency” MLS label that is distinct from that associated with other partitions. The TML attaches the “emergency” label to data it receives over the channel, restricting the data to only this partition of the E-Device. Data flows for emergency management are shown in Figure 6.3, as follows: (1) the authority propagates changes to the global emergency state and receives confirmation from the device; (2) the authority sends keys, policies, and revocations to the Emergency Management TSM, via an Extended Trusted Channel managed by the Trusted Channel Manager; (3) the authority sends encrypted emergency data to the emergency partition, via an Extended Trusted Channel; (4) data providers provide additional data for the authority to send to the E-Device; (5) when needed, the Emergency Management TSM decrypts the emergency data with keys and policies in its secure storage and provides access to the user within the emergency partition; (6) the Emergency Management TSM can send the most highly sensitive data to be displayed by the Trusted Path Application.

Trusted channels between the TML and the authority are protected using freshly negotiated channel secrets for each connection, based on the DRK rather than a stored root key. As a result, only parties with access to the DRK (the authority and the E-Device) can authorize an Extended Trusted Channel to the emergency partition.

Aside from these secure channels, the information cannot flow out of the emergency partition, e.g., to any other partition, device or network, ensuring that emergency data cannot be revealed outside of the equivalence class of components labeled as “Emergency”. Additionally, the emergency partition itself is only made available to the user by the TML and Trusted Path Application when an emergency has been declared, as previously described. This temporal restriction limits the threat of malicious insiders, since there is no access to emergency data most of the time. Additionally, the temporal restriction, in combination with the partition’s spatial separation, provides defense in depth for the confinement of emergency information.

The authority provides its own emergency data to the E-Device, and can convey data provided by third party data providers. When the latter data is proxied through the authority and the third party has no direct communication with the E-Device,
If a third party is considered trusted, it can be included in the “Emergency” equivalence class and allowed to establish an Extended Trusted Channel directly to the emergency partition. Since the third party does not have access to the E-Device’s DRK, it and the Emergency Management TSM share an “emergency” key, which is stored locally in the TSM’s persistent secure storage. Even when third party data is provided directly, the TSM on the E-Device can still be configured to only accept policies for that data directly from the authority.
All emergency data sent over the Extended Trusted Channel is encrypted by the authority or data provider prior to transmission using keys only available (on the E-Device) to the TSM. This enables the TSM to enforce its discretionary access control policy on use of the data by the responder or any software within the emergency partition, and audit the use of the data, even though it executes alongside untrusted software in the emergency partition.

**Trusted Application Display**

Within the emergency partition, the TSM can release data to untrusted, feature-rich commercial applications for display. However, there is no assurance that untrusted applications will accurately display data when asked. Some information, e.g., that which is critical and easy to manipulate, may require greater protection. For this purpose, we provide the high-integrity *Trusted Application Display* mechanism, which allows an application-TSM to send text-only emergency data directly to a reserved region of the display via a secure call to the TML that bypasses the untrusted software in the emergency partition.

The trusted display mechanism provides an unspoofable means for an application domain program to display messages with high integrity such that they cannot be observed or modified by any untrusted software in the system. This mechanism is available to the TSM in the emergency partition and the Trusted Path Application running in the trusted partition. Both the TSM and Trusted Path Application are designed to be evaluated for high assurance to ensure their correct behavior; this helps to ensure that the only legitimate data is provided as input to the trusted display mechanism, which will display it faithfully.

The TML virtualizes the video graphics card such that it appears to each partition that it has control of the screen. These virtual devices pass input to the TML’s secure display driver, which divides the physical display into two regions. One region is restricted for the TML-controlled high-integrity display (for example, the bottom two lines of text on the screen). The remaining region of the screen is exported to the partition with focus as usual.

High integrity data to be displayed is encrypted and either comes directly from the authority or is chosen to be released by a TSM during its operation. To pass the data securely to the TML, the Emergency Management TSM is divided into two pieces: an application-TSM in the partition and a kernel-TSM in the TML. The former is responsible for preparing the data for display, and the latter for passing the plaintext data to the TML securely. SP’s CEM protects the data as it is passed between privilege levels from the application-TSM to the kernel-TSM, through the untrusted software in the emergency partition, as described later in Section 6.4.1.

**Revocation**

To complete the data lifecycle, access to emergency data must be rescinded once the emergency is over. Data revocation takes place through complementary mechanisms, using both mandatory and discretionary access control.
The coarsest granularity of revocation available to the authority is to declare the emergency to have ended. As described in Section 6.3.3, this results in the closing of the emergency partition to users and applications, and restores its code and data to the pre-emergency state. Stopping application activity and overwriting the entire partition effectively removes all data generated or released inside the partition, regardless of how it was processed by untrusted software within the partition.

A finer-granularity of revocation is provided by the Emergency Management TSM itself, as described in Chapter 3. Over and above the TSM-enforced policies restricting access to data based on expiration dates, usage counts, search query restrictions, etc., at its discretion, the authority can communicate with the TSM and direct it to modify policies, keys, and other emergency restrictions to revoke access to existing data, for example, in preparation for ending the emergency. Usage and audit data can also be sent back to the authority at this time, after access to new data is cutoff but before the entire partition state is erased and reverted to a clean condition.

Guarantees to the authority and third parties about revocation and the state of the E-Device depend on connectivity and availability of the TSM. If the E-Device is disconnected temporarily from the network, or an application TSM managing communication with the authority is subject to a functional denial of service attack, the authority will be unable to synchronize the local emergency state with its own global state. An emergency expiration timer is provided by the TML, such that if connectivity with the authority cannot be established within a defined time, the TML can end the emergency on the E-Device. The use of this timer may not be appropriate for all responders and is therefore configurable.

Once communication is restored, the E-Device can attest to the authority that the requested updates to emergency state, policy, and keys have been made.

6.4 Interactions of Multiple TSMs

A traditional user-mode SP device or authority-mode SP device only supports a single trust domain for TSM code in the application space and only one CEM thread at a time, either executing or suspended, as described in Chapter 3. The limitation to a single trust domain is due to the fact that all TSM code shares access to the same DRK, SRH, and secure memory. All TSMs have the same access rights and are trusted to behave correctly with protected keys and data. As a result, the signature on the code only needs to distinguish the code’s memory location, and protected register state can safely be decrypted after an interrupt to signed TSM code in any process context with the correct virtual address. The limitation to a single CEM thread is due to the hardware support for interrupt protection and the fact that only one set of interrupt registers is provided on-chip to store the suspended CEM state.

Given only one trust domain, when a single application requires multiple TSM components, or multiple applications on the same device each need TSM code, all of the TSMs must be mutually trusting. They are signed and trusted by the authority and pre-verified to ensure they will behave correctly together. They must also be signed and loaded such that no two lines of code share the same virtual address, to
prevent code splicing between them. As long as these requirements are satisfied, TSM code can be located in any virtual address space and any process context. Data can be shared among multiple mutually trusting TSMs by sharing ciphertext in memory or on disk or by using CEM secure memory in a shared memory region. If the secure storage structure is shared amongst multiple TSMs, each TSM is trusted to enforce policies according to the same rules, which can include restrictions on all access to entire branches of the structure for certain TSMs.

For the E-Device and SecureCore designs, it is necessary to have multiple TSMs on a single device and to sometimes change these trust assumptions. In the simplest case, and for the E-Device as currently designed, there are mutually trusting TSM components in multiple privilege levels and for multiple purposes within certain privilege levels (e.g., TSMs in the TML as shown in Figure 6.3). These can operate largely independently, except that they must share the use of the SRH and coordinate to protect their respective storage trees; this is further described in Section 6.6.2.

The E-Device’s Emergency Management TSM also needs the ability to make TSM-to-TSM system calls, crossing privilege levels from the application in the user domain to the TML in the kernel domain. We enable this by creating a special TSM system call handler that can directly receive the call without leaving CEM, as described below.

When TSMs are required in multiple partitions on a SecureCore device, additional security mechanisms are necessary. The SecureCore TML guarantees strict separation between its partitions. While having mutually trusting TSMs in different partitions does not in itself create sharing between partitions, as the underlying data would be kept isolated by the separation kernel, the sharing of SP can create covert channels between the partitions. This stems from the fact that the SP instructions are not fully virtualizable, since the interrupt protection with suspended CEM state persists across partitions. We solve this problem in two parts. First, we provide a means for the TML or any hypervisor kernel to virtualize the basic CEM mechanisms and provide a distinct view of the CEM state to each partition or virtual machine. Second, we extend this to support all of authority-mode SP, where the SRH must be shared across the partitions to provide hardware-rooted integrity to each partition, without sacrificing the remote trust with the authority or the ability to have reliable revocation (i.e., transient trust).

6.4.1 TSMs in User and Kernel Domains

As previously described, the Emergency Management TSM has components in the application in the user domain (inside the emergency partition) and in the TML in the kernel domain. The application TSM needs to make direct calls to the kernel to access the high integrity trusted display. Any intermediate application code or guest OS code should not be able to intercept or modify the call, and secure data parameters and results need to be passed between the components. We can place a TSM in the TML kernel, however we must find a way to pass data between the two TSMs and to secure the call itself against interception by untrusted software.
Normally, TSM code cannot make any system calls to the operating system, since a system call would necessarily trigger an interrupt when calling into the kernel domain (and into non-TSM code), which encrypts all general registers. This prevents parameters from being passed in registers to the operating system interrupt handler, as is normally expected for system calls, and back from the interrupt handler for return values. Instead a TSM must exit CEM, have the unprotected part of the application make the call, and then return to TSM code. Thus the call, its parameters, and its return values are easily attacked or even blocked entirely.

In this case, we take advantage of having a TSM in both the calling application and in the kernel. We place the kernel TSM in a special system call handler to receive this TSM-to-TSM system call. Since TSM code is identified only by Begin_CEM and End_CEM instructions, the hardware makes no distinction between multiple blocks of TSM code within the same application or between those in different contexts. All TSM code can generate the same DRK-derived keys, read and write the same SRH register, and access secure memory regions. Thus parameters and return values can be passed in secure memory, which is protected by CEM hardware, or can be encrypted with DRK-derived keys and passed in unprotected memory. The only requirement is that the memory region containing the ciphertext must be available in virtual memory to the TSMs in both the user and kernel contexts.

To ensure the reliability of the call, the processor needs to stay in CEM continuously as the transition is made between privilege levels, since this would prevent any untrusted software from executing to intercept the call or to corrupt its execution state and cause a denial of service. As the typical system call path is triggered through a software interrupt and leads to non-TSM code in either the guest OS or the TML interrupt handler, this direct call is not normally possible. To address this, on an x86 platform, we setup a call-gate explicitly for calls to the kernel-TSM. The handler for this call-gate is itself TSM code, and since it is called directly from TSM code and is not the result of a hardware interrupt, the processor will remain in CEM across the call and privilege level change. The kernel-TSM can then itself exit CEM to make calls to non-TSM kernel components. It must later re-enter a TSM before returning to the user-domain context, which is expecting to resume execution still in CEM. This method also permits us to use general purpose registers to pass parameters and return values, as the registers will remain protected by the hardware while in CEM, even across the privilege-level change.

With this new mechanism for crossing privilege levels in CEM, the process for the application TSM to use the trusted display is as follows. The application TSM first decrypts the data using keys in its secure local storage and then stores the resulting display text in a memory buffer (at a known location) using Secure_Store instructions. This data is now only accessible in plaintext to TSM code. An x86 call-gate is used to transition from the application TSM to the kernel TSM without an interrupt. The kernel TSM uses Secure_Load instructions to read from the CEM-protected memory.

\textsuperscript{9}Other non-TSM software might attempt to also use this kernel-TSM call gate. In that case, the processor would enter the kernel-TSM routine but not in CEM, thus eventually causing a fault when a TSM-only instruction or TSM-signature MAC is encountered.
buffer and regular Store instructions to write the cleartext data to a TML buffer. It then exits CEM and invokes a TML-provided function for writing to the restricted region of the display. This TML routine will clear all plaintext copies of the data from TML memory after being displayed. Finally, the kernel TSM code re-enters CEM and returns to the call-gate in the application TSM, with a return value indicating the success or failure of the display operation.

6.4.2 Virtualization of SP

The builder of a SecureCore device can integrate the SecureCore software stack with either the user-mode or authority-mode SP architecture. The software on such a device would provide separate partitions for activities and data at multiple security levels, providing strict separation between them. For example, a device on a military network might have partitions for Top Secret, Secret, Confidential, and Unclassified data access, with separate software and communications options in each partition. The user could operate in any one of the partitions at a time, but could only move data between them according to MLS policies. This permits the local user to have a single device that can access all four levels of data, rather than requiring separate physical computers.

The addition of user-mode SP would allow local users to access their own personal secrets and credentials on the device, while providing portability of those secrets to other devices they also use. The users have such secrets on separate keychains for each of the security levels. Each partition has a user-mode TSM, signed for that device using its Device Master Key (DMK). Since the SecureCore least-privilege separation kernel enforces MAC policy, these TSMs need not be verified for MAC policy enforcement, but instead for the protection of keys and the containment of user-owned data within the partition whose guest OS and other software may not be trusted.

To enable such usage scenarios, where there are TSMs in more than one partition, the SP architecture must be virtualized. In order to accomplish this, in Chapter 7 we describe how to virtualize Concealed Execution Mode used by both user-mode and authority-mode SP, and the special features needed for remote trust that are unique to authority-mode SP. We also discuss the implications that SP introduces for preventing covert channels between partitions, a critical security feature for SecureCore.

Once SP virtualization is added and TSMs can be run in more than one partition, it is becomes possible for an E-device to have multiple emergency partitions for different purposes. Each of these partitions can maintain a different emergency state with the authority and contain different data. This might be useful if different potential emergencies might require data at different security levels, or to separate emergency data from mutually distrusting parties.

Integration with the SecureCore software stack also provides a possible solution to the trusted I/O requirement imposed by user-mode SP for entering the User Master Key, as the trusted partition can reliably accept this input without risk of attack from the untrusted software.
6.5 Security Analysis

6.5.1 Protection of Emergency Data

Secure Communications Channels  The basic Trusted Channel provided by the TML ensures that no party on the untrusted network can violate the confidentiality or integrity of the communications, using standard protocols. Furthermore, the TML authenticates the endpoints of the communication, preventing man-in-the-middle attacks but also verifying the security level of the endpoints. Thus, for example, if the TML will extend a trusted channel to the emergency partition, it must first authenticate the remote party and authorize them to receive and transmit data tagged with the “Emergency” label. The remote party is therefore considered to be trustworthy at that security level and is permitted by the access control policy to have read/write access to all data in the partition.

TSM-TSM Channels further encrypt and MAC their communications using DRK-derived keys. Thus the ciphertext data may be shared freely within the partition and with all remote parties at the same security level, however only another TSM on the same device or at the remote authority that shares the DRK can access the data. An underlying TML-provided channel is the only means for a TSM to share any data, encrypted or not, outside of the partition. Thus, a TSM cannot even share its ciphertext data with another TSM on the same device unless the data flow is permitted by the TML’s MAC policy. Similarly, the TML has control over the use of all I/O devices and only creates flows that are permitted by the policy.

In implementing these channels, the Trusted Channel Manager is trusted as part of the TML to make decisions related to MAC policy enforcement. Its TSM component, however, receives extra protection — ensuring that, for example, the channel keys cannot be obtained by an adversary when the device is offline, preventing the adversary from using the keys to impersonate a legitimate device. This protection allows a remote party to trust that data it receives from the device, tagged at a particular security level, is high-integrity data that did not come from an adversary. When the channel keys are cryptographically tied to DRK-derived keys, the remote party can be further assured that its channel is with a legitimate E-Device that is still under the control of the authority.

The remaining information flow is to the local authenticated users, who are trusted not to intentionally distribute data displayed to them when they are given a clearance for a particular security level. Such “Trusted Application Display” flows go through the Trusted Executive, which is verified and trusted not to create any covert channels between security levels or inappropriately share data it is given, and will ensure that data is reliably displayed to the users.

Both a TSM-TSM channel and a Trusted Application Display channel depend on untrusted code inside the emergency partition to schedule the TSM and reliably pass encrypted data through to the TML. This means that the untrusted software could launch a denial of service attack, but cannot breach confidentiality and integrity of the data.
Emergency State Management  Emergency state management message generation is a security critical operation. Only the authority is able to generate valid messages for a given E-Device as they are based on the device-specific DRK, known only to the authority and the E-Device. Since SP hardware ensures software never has access to the DRK and the authority secures its copy of DRK, arbitrary parties cannot generate a valid message. Since the emergency state change generates a response that is also cryptographically signed by a DRK-derived key, the authority can be assured that the emergency state management was correctly processed by the intended E-Device. State change messages are also confidential since they are encrypted with DRK-derived keys. Even the TML cannot itself view the message directly; it must first pass the state change message to the E-State TSM to be processed, and then query the TSM for the new state.

There are several layers of protection that detect and prevent replay attacks on emergency state management messages: (a) all messages are transmitted over secure primary channels such that only the subjects with access to the endpoints of the secure channels (e.g., the TML) will even know that a state management message was transmitted and be able to see the encrypted message; (b) the message is encrypted using keys derived from the DRK, such that only the TSM on the correct E-Device can decrypt and process the message; (c) the monotonically increasing counter ensures that a given message is never processed twice.

Containment of Emergency Data  Once an emergency is declared and the E-Device successfully changes its emergency state, the emergency partition is enabled and the user is able to access it. The TML ensures that the emergency partition can write only to channels leading to the authority and to trusted data providers. Further, no other partition on the E-Device can read content in the emergency partition labeled as “Emergency”. The TML provides virtual device abstractions for only trusted physical devices to the OS in the emergency partition. The combination of these restrictions ensures that no information can flow out of the partition to other local or remote devices and networks except as defined by the MAC policy, therefore containing emergency data only to entities that share the “Emergency” security label.

When the emergency data is decrypted and displayed within the emergency partition, the untrusted applications or OS may keep parts of the plaintext emergency data in memory or write it to disk, but the TML’s separation policy ensures that the data remains within the containment of the emergency partition. While all data in memory is erased or otherwise invalidated on a shutdown of the E-Device, the data on disk will remain accessible if the emergency partition is still active. Any offline attacks on the emergency data on disk are prevented, as the TML protects all data on disk by encrypting each partition’s virtual disks with keys derived from the DRK. These encryption keys are derived as needed and are never revealed or stored outside of a TSM. Thus the TML can ensure emergency data containment for as long as the emergency state and partition remain active. When the declaration of the emergency is rescinded, the emergency partition becomes inaccessible to the user and its contents are encrypted and stored for audit purposes or are immediately deleted.
Applications in the emergency partition are not expected to display high integrity content, as both the applications and the guest OS are not trusted and could modify content before it reached the display. Instead, high integrity information is displayed on the reserved portion of the screen, via the Trusted Application Display, with data that is appropriately encrypted and hashed by the TSM. Since the TML manages the physical display, no partitions are given direct access to the portion of the screen reserved for high integrity display.

6.5.2 Multiple TSMs

When there are multiple TSMs on a single SP device for either authority mode or user mode, all TSMs of the same type must be mutually trusting. Software can make calls to any TSM code that it can read in memory or on disk, and all TSMs share access to the same DRK-derived keys and secure memory. Therefore all TSM code on the device should be aware of data stored and used by other TSM components and correctly enforce access control for that data. The separation kernel can prevent a TSM from using or releasing data by controlling access to stored ciphertext, but the TSM should not rely on protection mechanisms of non-TSM code for its own policy enforcement. Similarly, if TSMs are placed in multiple virtual machines on the same device, the code will all be signed with the same device key, and will not be limited by the hardware to its designated virtual machine. Therefore the TSMs must be written to enforce access control as if all TSM code could be used from any virtual machine.

When data is passed from one block of TSM code to another, such as a user-domain TSM passing data to a kernel-domain TSM, the data should be placed in registers (which are only accessible to TSMs while in CEM) and secure memory, or should be encrypted and MACed with DRK-derived keys, to prevent untrusted software from reading or modifying the data. Even when using TSM-TSM call gates to directly call a kernel-TSM from the user-TSM, an interrupt could still occur any time during the operation, permitting non-TSM software to regain control of the CPU and access unprotected memory. All registers and secure memory will be protected while CEM is interrupted, and the TSM will resume where it left off when the process is resumed.

Without the call gates, the untrusted software would need to intercede and relay all kernel calls on behalf of the TSM, and then call back into the TSM after each call is complete. This would provide an opportunity to deny service, skip an intended call, fake a call, or replay data. While the TSM could attempt to detect many of these attacks, it is still dependent on other software to make the call. When the call gates are used from within the TSM, the untrusted code can only deny service to the TSM in its entirety; if it runs the TSM code at all, it cannot prevent the kernel-call from being made. It can only use interrupts to suspend the TSM, but cannot change the execution flow to prevent a call altogether.
6.6 Implementation of SecureCore Prototype

To demonstrate the functionality of our E-device and the feasibility of writing an Emergency Management TSM, we have created a prototype implementation\(^{10}\) for some of the key features of our system [47]. Our prototype includes:

- an Emergency Management TSM maintaining emergency state and implementing discretionary access control policy on emergency keys and data;
- an interface for this TSM to interact with untrusted software in the emergency partition;
- an interface to accept data from the authority and send responses;
- a simulated authority system, maintaining its own copies of the DRK to sign and verify messages when communicating with the TSM;
- a prototype of the layered SecureCore software stack (taken from [104]) to demonstrate the control flow, data and MAC policy interfaces, and virtualization provided by the TML; and
- an emulation module that implements the SP hardware features below the TML.

The prototype integrates all of these components and the communication between them to allow emergency messages and data to be stored, communicated, and accessed by a user of the E-Device.

Using this prototype, we have been able to demonstrate key management and emergency state management by the authority along with valid local accesses to emergency data. Data is presented to the user through the application in the emergency partition and using simulated display through the trusted path.

We further demonstrate and verify the security provided by the TSM. First we have tested denial of access due to policy enforcement on keys and keychains, with authorization dependent on emergency state. Second we have observed the detection of attacks on the communication links between trusted parties and on the local storage of secure persistent data.

6.6.1 Prototype Architecture

Figure 6.4 shows the architecture implemented\(^{11}\) in the prototype on the left in comparison with the real hardware design of the SecureCore architecture on the right. In the prototype, only the emergency partition is implemented, running the SecureCore OS — an implementation of a Trusted Executive — in place of an untrusted guest OS.

\(^{10}\)This implementation was joint work between Princeton University, the Naval Postgraduate School, and the USC Information Sciences Institute (the authors of [104]).

\(^{11}\)The TML and SecureCore OS were implemented by the Naval Postgraduate School. The SP Emulation Module was implemented by Jeffrey S. Dwoskin (Princeton). The Emergency Management TSM, emulated authority, and other application components were implemented by Jeffrey S. Dwoskin (Princeton) and Ghanesha Bhaskara (USC Information Sciences Institute).
The TML hosts the SP Emulation Module to provide SP instructions to the TSM; an interface to the SP Emulation Module is provided by the SecureCore OS to pass through this emulation library across the guest OS privilege level. The SecureCore OS also provides interfaces for access to the display, keyboard input, and local user authentication. The real TML would support SP virtualization as described in Section 6.4.2 and would be otherwise unaware of application-TSM access to SP hardware features. In both cases, the Emergency Management TSM is implemented as user-level code and provides services to the untrusted application through the TSM Interface that is described in Table 6.1.

Our prototype TSM handles emergency state management in the application layer, rather than integrating a second TSM into the TML. Additionally, output to the trusted display is simulated in the TSM since no trusted partition is available on the prototype to host this feature separately.

CEM protection on the prototype is simulated by the SP Emulation Module and does not protect the TSM against all attacks on its execution. The prototype of the application, TSM components, and authority could instead be implemented on the
testing framework described in Chapter 5 for more protection and to permit testing of attacks on CEM itself.

A goal of the SecureCore project was to create a clean-slate design for a trusted software stack and secure hardware containing roots of trust. Therefore, our prototype was designed to demonstrate each of the key components in this new architecture, in particular the interactions between components at each layer of hardware and software.

6.6.2 Key Management and Secure Storage

The Emergency Management TSM maintains a number of keychains, each of which contains one or more keys along with the policies that apply to those keys. Each keychain can be enabled or disabled by the authority, and can be limited to be available to the local user only when the emergency state on the device is above a preselected level. Data can then be encrypted and MACed with the keys and stored on the device in encrypted form. The user’s application can make calls to the TSM to access this data whenever the keychain is enabled. The per-key policies currently allow for use to be further limited to a particular authenticated user and limited to a set number of uses, demonstrating automatic expiration.

Only the authority can add or delete keychains, and does so by sending messages to the TSM that are signed with DRK-derived keys. As it adds a keychain, the authority provides an encryption key and a MACing key that are used to send further signed keychain-management commands, such as adding and deleting keys and policies. By keeping those keys secret or by sharing them with a third party, the authority can restrict access to the keychain or delegate access to a third party. If the authority wants to specify policies for data that is to be sent later by a third party, it maintains the keychain-management keys itself but shares the keys on the keychain with the third party to encrypt and MAC data; the third party then cannot send keychain-management commands that modify the keys or policies.

Figure 6.5 depicts the prototype implementation of the secure local storage of the TSMs on the E-Device. It follows the tree configuration of Chapter 3, but uses a fixed node structure. At the first level, each sub-tree represents the data needed by each of the TSM components (from Figure 6.3). The keychains belong to the Emergency Management TSM to store the keys, policies and data used during an emergency. Additionally, the E-State TSM stores the current emergency state, emergency level, and emergency transition counter. Finally, the Trusted Channel Manager TSM stores keys and state used to establish secure communications channels.

For the prototype (Figure 6.4), the functionality of the E-state TSM has been incorporated into the Emergency Management TSM. The Trusted Channel Manager and its TSM have not been implemented, since the TML’s MAC policy enforcement is not part of this prototype.

In a system where multiple TSMs are implemented in a non-virtualized SP architecture, they will all share access to the SRH register. Since there is then only one SRH value to root the persistent storage, all TSMs must share the same secure storage structure and are trusted to access it correctly. This means that at a minimum,
any TSM may need to verify the first level of the tree and regenerate the on-chip SRH value when changes are made to any one branch. Access to the ciphertext of each of the encrypted sub-trees of the secure local storage may be limited to only the appropriate TSM, as long as a current value of the MAC of the sub-tree’s root node is available to all TSMs.

### 6.6.3 TSM Interface

The Emergency Management TSM provides an interface to both the remote authority, using signed messages sent over the network, and to the local user through an API. This interface is shown in Table 6.1. The user accesses the TSM through the untrusted Emergency Management application, which accesses the TSM’s API through a set of library calls. The application also receives management commands over the network from the authority and third party data providers as signed messages and passes them through to the TSM library, presumably unmodified.

When such a management message is received, the application calls the first library function, `TSMLib_Process_ExternalMsg()`, to submit the message to the TSM. The message specifies one of the commands listed in Table 6.1, along with the appropriate parameters. For emergency-state and keychain-management commands, the
encryption and signing keys used to authenticate the message are DRK-derived keys, and therefore only the authority can issue these commands. This message includes two nonces in plaintext that are used within the TSM to derive the keys to verify the integrity of the message and then to decrypt the rest of its contents.

When the authority creates a keychain, it provides new encryption and MAC keys that are stored with the keychain and used to generate and verify messages for key-management on that keychain. Thus, as described previously, the authority can authorize a third party to manage the keys and policies on a keychain by sharing those keys; the third party will not have nor need any access to the DRK — the assigned keys are sufficient to send TSM_Key_Add and TSM_Key_Delete commands.

The emergency state management functions, Update_State and Get_State, are implemented inside the TSM. When TSM_Set_Emergency_Level is used by the authority, the TSM internally calls Update_State to set the binary emergency state to “on” for any emergency level greater than zero, and “off” otherwise. TSM_LIB_Get_Emergency_Level() can be called to get access to Get_State, but may be limited to the TML and thus not available to untrusted applications.

As the authority creates a keychain with the TSM_Keychain_Create command, it specify what emergency levels are necessary to enable access to the keychain. It can also use TSM_Keychain_Enable and TSM_Keychain_Disable to turn entire on and off access to entire keychains regardless of the emergency level and TSM_Keychain_Delete to permanently delete a keychain.

The next set of TSM library functions are key operations (TSM Lib KeyOp*( )); they use keys on the keychain to perform cryptographic operations. Before performing any operation, the TSM first checks the keychain’s policies (based on the current emergency level) and the per-key policies. The per-key policy may restrict access to a particular authenticated user. User authentication is provided by the TML and is passed in to the TSM using a signed user authentication token. This token is generated by the TML when the user authenticates through the Trusted Path Application (or through the Trusted Executive for the prototype); the TML and the TSM will have a pre-shared key to generate and verify the tokens. If a per-key policy specifies usage limits, the TSM will record each access to the key and write-back the secure storage with the new usage data, eventually cutting off access according to the policy. The TSM permits different policies to be set for each type of operation on a key, such that keys can be designated for certain purposes, such as for encryption-only. Another useful policy is “re-encryption”, where a particular key can only be used to decrypt data if it is then re-encrypted within the TSM using another key that permits encryption. Therefore the associated data will never leave the TSM’s protected environment in plaintext. Keys can similarly be designated to be used to generate or verify a MAC or to generate a new software-derived session key.

The last set of TSM library functions control reading and writing the secure storage to and from persistent storage. Since the TSM cannot make system calls, it relies on the application to load and store the ciphertext data on disk. The application loads the secure storage data into memory and then calls TSM Lib Load SecureStorage() with a pointer to the structure. The TSM will then verify and decrypt the data, storing it in secure memory. The TSM can perform operations and update the struc-
ture within secure memory, where CEM protection prevents any other software from accessing or modifying the data. When it is ready to commit data, the TSM will encrypt the new structure and update the on-chip SRH value with a new MAC. Then it relies on the application to write the new ciphertext to disk. The application calls `TSMLibGetSize_SecureStorage()` to determine how much space is needed and allocates a region of memory to hold the ciphertext. This may be different from the space originally used to load the structure. Then the application calls `TSM-Lib_Store_SecureStorage()`, which causes the TSM to copy the ciphertext back to the applications memory region from its secure memory.

Table 6.1: Emergency Management TSM Interface for Local User and Remote Authority in the SecureCore Prototype.

<table>
<thead>
<tr>
<th>TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSMLib_Process_ExternalMsg (tsm_msg_t *message, size_t size, counter_t counter, msg_sig_t MAC)</td>
<td>Process a remote command from the authority or a third party to create/delete keys or keychains. The message structure specifies the command and its parameters as well as any nonces used to generate keys that were used to encrypt and sign the message. The counter prevents replay, and the MAC verifies the integrity and source of the message. The authority uses DRK-derived keys to encrypt &amp; sign messages to create/delete keychains or to enable/disable keychains. The authority and third parties use dedicated keys to add/delete keys on keychains they have authorization for.</td>
</tr>
</tbody>
</table>

(Continued on next page)
### Table 6.1: (continued)

<table>
<thead>
<tr>
<th>TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM_Set_Emergency_Level (level_t emergency_level, unsigned int emergency_counter)</td>
<td>Authority sets the emergency level and emergency state of the device, to either EMERGENCY_NONE (0) for no emergency or a value in the range from EMERGENCY_LOW (1) to EMERGENCY_HIGH ( n ) for an active emergency. The new counter value is set and must be monotonically increasing from the previous value. The message is authenticated using nonces and DRK-derived keys.</td>
</tr>
<tr>
<td>TSM_Keychain_Create (keychain_t keychain_ID, key_struct_t keys, level_t emergency_level)</td>
<td>Authority creates a new keychain, specifying the new ID and the key structure (encryption key, MAC key) that will later authenticate access to the keychain. The emergency_level specifies the minimum emergency level that must be set on the device in order to use the keys on this keychain; this is saved in the keychain policy. The message is authenticated using nonces and DRK-derived keys.</td>
</tr>
<tr>
<td>TSM_Keychain_Delete (keychain_t keychain_ID)</td>
<td>Authority deletes an existing keychain entirely, specifying its ID. The message is authenticated using nonces and DRK-derived keys.</td>
</tr>
<tr>
<td>TSM_Keychain_Disable (keychain_t keychain_ID)</td>
<td>Authority disables the use of an entire keychain, regardless of any policies attached to the keys. The message is authenticated using nonces and DRK-derived keys.</td>
</tr>
<tr>
<td>TSM_Keychain_Enable (keychain_t keychain_ID)</td>
<td>Authority enables the use of a keychain that has been previously disabled. The message is authenticated using nonces and DRK-derived keys.</td>
</tr>
<tr>
<td>TSM_Key_Add (keychain_t keychain_ID, key_struct_t keys)</td>
<td>Add a new key structure onto an existing keychain (includes key and policy). The message is authenticated using the keychain-specific encryption and MAC keys.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 6.1: (continued)

<table>
<thead>
<tr>
<th>TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSM_Key_Delete (keychain_t keychain_ID, key_t key_ID)</td>
<td>Delete an existing key from a keychain, specifying its ID. The message is authenticated using the keychain-specific encryption and MAC keys.</td>
</tr>
<tr>
<td>level_t TSMLib_Get_Emergency_Level (user_t user)</td>
<td>Retrieves the current value of the emergency level and state of the device. May be optionally limited to the TML only or require user authentication.</td>
</tr>
<tr>
<td>TSMLib_KeyOp_Encrypt (user_t user, keychain_t keychain_ID, key_t key_ID, void *data, size_t len)</td>
<td>Encrypt data with the specified key.</td>
</tr>
<tr>
<td>TSMLib_KeyOp_Decrypt (user_t user, keychain_t keychain_ID, key_t key_ID, void *data, size_t len)</td>
<td>Decrypt data with the specified key.</td>
</tr>
<tr>
<td>TSMLib_KeyOp_ReEncrypt (user_t user, keychain_t keychain_ID, key_t dec_key_ID, key_t enc_key_ID, void *data, size_t len)</td>
<td>Decrypt data with one key and encrypt with another key from the same keychain in one atomic operation. Requires permissions for “re-encrypt” on the decryption key and “encrypt” on the encryption key.</td>
</tr>
<tr>
<td>TSMLib_KeyOp_GenerateMAC (user_t user, keychain_t keychain_ID, key_t key_ID, void *mac, void *data, size_t len)</td>
<td>Generate a MAC over the data with the specified key.</td>
</tr>
<tr>
<td>int TSMLib_KeyOp_VerifyMAC (user_t user, keychain_t keychain_ID, key_t key_ID, void *mac, void *data, size_t len)</td>
<td>Verify an existing MAC over the data with the specified key. Returns a result for either a “MAC Match” or a “MAC Mismatch”</td>
</tr>
<tr>
<td>TSMLib_KeyOp_GenerateSessionKey (user_t user, keychain_t keychain_ID, key_t key_ID, void *sessionkey, void *nonce, size_t len)</td>
<td>Generate a new session key using the specified key and provided nonce data.</td>
</tr>
<tr>
<td>TSMLib_Load_SecureStorage (void *addr, size_t size)</td>
<td>Tells the TSM where the encrypted secure storage structure is located in memory.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 6.1: (continued)

<table>
<thead>
<tr>
<th>TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t TSMLibGetSizeSecureStorage (void)</td>
<td>Returns the amount of memory necessary to save a copy of the current secure storage structure. Used by the calling application to pre-allocate memory before calling TSM-Lib_Store_SecureStorage().</td>
</tr>
<tr>
<td>TSMLib_Store_SecureStorage (void *addr, size_t size)</td>
<td>Requests that the TSM write an updated encrypted copy of the secure storage structure to a memory location to be saved to disk. The TSM will also update the SRH register.</td>
</tr>
</tbody>
</table>

6.6.4 Lessons Learned

During the design phase of the prototype and its implementation, we were able to gain insights about how to design the emergency device, impacting each system level: SP hardware, the hypervisor, the TSM, and the authority. The SecureCore architecture and certain aspects of the SP architecture represent the results of applying these lessons. We discuss some of the lessons in detail below.

**Design** One early challenge in the design phase of the SecureCore prototype was to determine how to organize the secure storage structure and what kinds of policies are necessary for emergency management. Each type of policy requires meta-data, policy data structures in secure storage, enforcement mechanisms in the TSM, and new interactions between the authority and the TSM for setting the policies. When choosing access control policy mechanisms it is especially important to consider how TSM code can perform the access control check and what types of support it will need from other system components. To implement user-specific policies, we adopted a requirement that the TML generate secure tokens for user authentication, which are passed into the TSM when accesses are made. A complete implementation would additionally benefit from a secure time source. TSMs could also support more robust policies if system software provided the TSM with identification of the application that is calling the TSM library when access to data is requested.

When designing the full SecureCore and prototype architectures, we determined which components in the system needed to have TSM support and which could rely purely on TML software to provide the security guarantees in our threat model. Throughout the design process, we developed and tweaked the responsibilities of trusted software and how these were to be subdivided into TSM components at different system levels. Many of the decisions were made based on reliability requirements (pushing certain components into the TML) and on the need of the TML to
guarantee enforcement of its MAC policies, even if the TSM’s security model would be unaffected.

For example, both the TSM and the TML setup secure communication channels with the authority and third party data providers; these channels have different endpoints within the software layers, depending on the intended use of the channel, and rely on different sources of cryptographic keys to secure the communications. The two sources of secure channels had to be integrated, using a design that evolved over time such that TSM-TSM channels could be tunneled through other types of trusted channels. At the same time, the Trusted Channel Manager relies on a TSM of its own to protect the keys necessary to create basic trusted channels. Similarly, emergency state management messages are passed through multiple channels to be verified by the TSM inside the emergency partition, and new emergency data is passed through trusted channels into the emergency partition without violating the TML’s MAC policies.

Implementation During the implementation, we had to determine how to integrate authority-controlled keychains and policies with third-party controlled keychains. It was not clear how much control the authority should be able to delegate to third parties, how to do so securely, how the authority could revoke access and enforce higher level policies, and at what point in the creation of a keychain the control should be delegated. We found hybrid policy control to be the most effective and flexible; the authority must create each keychain and can set policies that cannot be overridden, and the third party is permitted to add and delete individual keys that can contain additional policies. Ultimately the authority maintains the ability to enable, disable, or revoke entire keychains.

The implementation also forced us to carefully consider how the TSM interacts with other software — both the hypervisor and the untrusted applications. While the original SP architectures did not allow any software to manipulate CEM state, the hypervisor needs this control for virtualizing the hardware simply to maintain its security guarantees. The exercise of designing the interface for the hypervisor with TSM code and with SP hardware respectively, led to the architectural changes in Section 6.4.1 and Section 7.2.

Regarding the untrusted applications that call a TSM, we had to address how the TSM could load and store the persistent secure storage; this process involves loading an arbitrarily large structure from the filesystem, allocating both regular and secure memory, transferring data between the two types of memory, checking and setting the SRH register to maintain the integrity of the structure as changes are made, and writing back the updated structure that may have changed size. Since the TSM code cannot make system calls, it must rely on the untrusted application to interact with the OS for many of these operations. The process is further complicated by the fact that the interaction is one-way; the application makes calls to the TSM API, but the TSM does not make calls back into the application. Implementing each of these operations necessitated modifications to the TSM API and careful security analysis to ensure that a malicious application could not cause the TSM to violate its
confidentiality and integrity guarantees. Designers must consider time-of-check-to-time-of-use attacks and that the various TSM functions might be called out of order, or not called at all.

Another challenge was for the initialization routines in each block of TSM code. For efficiency, a TSM loads the persistent secure storage from disk when it is first called and keeps it loaded in secure memory for subsequent calls. Therefore each time a TSM function starts with a Begin_CEM instruction, it must verify that the secure memory is initialized in the hardware and check if the secure storage structure is already loaded and verified against the SRH. If the secure memory is not initialized, the application and the OS could have written arbitrary data to the TSM’s regular non-secure memory. Consequently the TSM cannot safely use regular memory to keep track of the state of the secure memory regions. Earlier versions of the SP instruction set for secure memory did not provide a straightforward and secure way for the TSM to check if a particular region of memory was already protected as secure memory, which we have added for Secure Areas. This is necessary to see if the data stored in memory was in fact loaded by a previous TSM and can be trusted.

These software design challenges in the TSM also had implications for how the TSM and the authority must interact. The TSM establishes secure communications with the authority and allows the authority to make changes to the secure storage structure (e.g., updates to the policies or emergency state). However, these changes are not permanent until the TSM writes the updated secure storage structure to disk in encrypted form and updates the SRH register on-chip. When designing processes that depend upon both trusted and untrusted software, the sequence of the steps is critical. In this case, the TSM must not report success of the changes to the authority until the new structure is written to disk and the new SRH value is set in hardware. Furthermore, the secure storage must not be left in an inconsistent state if the process is disrupted by a power failure or attack. The new storage structure should be written, the SRH updated, and then the old storage structure deleted. This way if a disruption occurs at any point, the SRH value on chip corresponds to a version of the structure that is still available on the device.

6.7 Summary

In this chapter, we explored the implications of integrating authority-mode SP with trustworthy system software — trusted hypervisors, separation kernels, or trusted operating systems. The integration permits improved usability for the local user and stronger guarantees limiting dissemination of emergency data for the authority and third party data providers.

As part of the SecureCore project, we integrate SP with the SecureCore software stack and demonstrate that SP is compatible with mandatory access control systems that provide strict separation of multiple security levels on a single SP device without introducing covert channels. We show how SP can be used in the Trusted Management Layer to enhance the security of the remote management of emergency state and of secure communication channels. These new uses allow us to examine how multiple
TSMs can interact on a single device between different privilege levels and security contexts provided by the system software.

Finally, we document our prototype implementation of the Emergency Management TSM. This prototype provides for interactions between the authority, TSM, emergency management application, third party data providers, and local user through an API. We provide the layout of the secure local storage used by the TSM to store emergency state and emergency data, and show how it can reliably implement transient trust for the authority, storing access control policies and managing keychains for both the third party data providers and the authority. We demonstrate how the prototype architecture fits into the full design of the SecureCore architecture, which provides robust protection of emergency data on a remote device. By designing this prototype in detail, we hope to show that it is reasonable to partition trust on an Emergency Device with the highest priority secrets entrusted only to a TSM, and with additional guarantees made by the system software.
Chapter 7

Virtualization of SP and Dual-mode SP Devices

7.1 Introduction

In this chapter, we discuss two improvements to the SP architecture. First, we have designed new features to virtualize the SP architecture — both for the basic Concealed Execution Mode and for the authority-mode remote trust capabilities.

Second, we consider how to make a single SP device that supports both user-mode and authority-mode TSMs, to allow users more secure personal use of their devices in non-emergency situations. This is done without compromising the security properties that the authority relies upon, and simultaneously provides a solution to an attack on user-mode SP.

7.1.1 User-mode SP Architecture

As described in Section 3.7, the user-mode SP architecture [102] was designed to protect users’ personal keys, on one or more keychains, from the untrusted software on their devices. They will install their own trusted software for Concealed Execution Mode on any SP devices they owns to access their keys and make use of the data that the keys protect. If a device is compromised, their keys will remain secure under the control of their TSM.

A user-mode SP device has two hardware roots of trust, the Device Master Key (DMK) and the User Master Key (UMK). The DMK is the device key and serves most of the same purposes as the Device Root Key on an authority-mode SP device; it is stored in a non-volatile register on-chip and is used to sign user-mode TSM code (during initialization), to encrypt general registers during an interrupt, and to protect secure memory. The UMK is volatile and is set during each session through a secure input channel from the user. The user enters a passphrase into the secure input mechanism, which is used by the hardware to derive a key that is stored in the UMK register. The UMK register can be read by user-mode TSM code and is used
as the root key to encrypt and MAC the user’s keychain.\textsuperscript{1} A user-mode SP device does not have an SRH register and does not allow the TSM to derive keys from the DMK. Therefore it has no device-specific storage or reliable revocation mechanisms — all user keychains can be transferred between any of the user’s devices.

\section{Virtualization of SP}

\subsection{Virtualization of Concealed Execution Mode}

In Section 6.4.2, we developed a few usage scenarios for SecureCore where it is desirable to have TSMs in multiple virtual machines on a single device, using either user-mode or authority-mode SP. Generally each of these TSMs would access different sets of sensitive data or different keychains. In this section, we will discuss how to virtualize SP’s Concealed Execution Mode to permit TSMs in each virtual machine. In the following section, we build on this solution to fully virtualize authority-mode SP, addressing the additional complications of sharing the on-chip SRH and providing both virtualization and remote trust together.

As discussed in Section 6.4, there is no fundamental problem with having multiple TSMs on a single device. The hypervisor kernel will provide each virtual machine (or partition) with isolated storage and processor resources, preventing sharing of ciphertext on disk, secure memory data in cache or main memory, and intermediate data in registers. The complication is that there is only one set of on-chip registers for the CEM state and CEM interrupt data (see Figure 3.3). Sharing these registers and providing access to them from all virtual machines (VMs) via the SP instruction set creates a storage covert channel \cite{99, 121, 184} between the machines, violating the isolation properties of the SecureCore software stack. If one VM has a suspended CEM thread, this blocks any other TSM from starting in the same or a different VM. This passes at least one bit of information from one VM to another — whether or not another VM has a suspended CEM thread.

Another potential storage channel in user-mode SP is the User Master Key register itself. If this could be set by software in one VM and then read from another, then a channel would exist. Fortunately, we do not permit this register to be set directly by software, but rather only through the trusted I/O path. In addition, we expect the value to be user-specific and shared for their TSMs in all VMs, rather than entering separate values for each security level. Similarly, if software in one VM were to overwrite the Device Master Key, then the inability to execute a TSM in another VM would reveal information. We assume some hardware restriction on when the Device Master Key can be written to prevent such an attack, but additionally realize that once the Device Master Key is modified, it cannot be changed back to the original value to send additional data. Thus such a channel would only provide 1-bit total, rather than 1-bit at a time as a covert channel.

\textsuperscript{1}A user can have multiple user-mode keychains, each rooted with a key derived from a different passphrase.
To prevent the covert channel created by the suspended CEM state, we permit the hypervisor (e.g., the TML) to virtualize the CEM state by swapping it out of the on-chip registers to protected kernel memory. Any time the hypervisor is going to switch VM contexts, it will save the SP CEM state for the outgoing VM and restore the SP CEM state for the next VM to execute. The saved state is composed of the CEM Status/Mode bit (CS) — either ‘Interrupted CEM’ or ‘Normal’\(^2\), the CEM Interrupt Address (CEM.IntAddr) along with any additional process ID (PID) and virtual machine ID (VMID) bits, and the CEM Interrupt Hash (CEM.IntHash). In order to protect the confidentiality and integrity guarantees of SP, the entire saved state will be encrypted and MACed with the DRK before providing the contents to the hypervisor. Thus the hypervisor will be unable to observe or modify the CEM state of any VM; it can only replay saved state between VMs or over time, which it is trusted not to do, and can cause a denial of service, which it can do regardless.

Table 7.1 shows the two new instructions, \textit{Save\_SPRegs} and \textit{Restore\_SPRegs}, that implement user-mode CEM virtualization. Only a trusted hypervisor, such as the TML, can execute these instructions. The hardware will restrict their execution based on privilege level or processor mode, depending on how hardware virtualization support is provided. Once executed, the operations are atomic; it must not be possible to interrupt the saving or restoring of SP state, as this could leave the processor in an inconsistent state with undefined results for CEM operation. When a virtual machine switch is triggered in the hypervisor, it must save the state of that virtual machine and load the state of another one. As part of this process, it saves the regular non-SP processor state and calls \textit{Save\_SPRegs} to store the SP state. The hypervisor maintains a structure in its protected memory with the saved SP state for each virtual machine and must protect this memory region from any guest operating systems or other untrusted software. Once the hypervisor has loaded the next virtual machine into the processor, it calls \textit{Restore\_SPRegs} with the previously saved SP state for that machine. It can then begin execution of this virtual machine. Additionally, in order to provide a mechanism for initializing the SP state of a new virtual machine that has no saved SP state, the \textit{Restore\_SPRegs} instruction accepts an SP state of all zeroes as a special case. It will bypass the hash verification of the saved data and instead setup a new empty SP state for the machine. The CEM Status will be “Normal” and the interrupt registers will be cleared, thus permitting a TSM to execute in this virtual machine.

The state transition diagram for user-mode SP virtualization is shown in Figure 7.1. The first three states are the same as without virtualization, with “Active” state for a TSM currently executing, “Interrupted CEM” state for a TSM that has been suspended, and “Normal” state when a TSM may be loaded with the \textit{Begin\_CEM} instruction and no other TSM is currently active or suspended in the processor. To this we add a new state, “CEM Saved”, to be used when the SP state for the current virtual machine has been saved by the hypervisor, but the next SP state has not

\(^2\)The saved SP state can never be ‘Active CEM’ since the save and restore operations for virtualization are not permitted while a TSM is executing. In order for a hypervisor to save the SP state, an interrupt or exception must occur first, causing the CEM Status to become ‘Interrupted CEM’ if in CEM at the time or ‘Normal’ if the TSM had previously exited on its own.

135
Table 7.1: New Instructions for Virtualization of Concealed Execution Mode

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save SPRegs Rs,imm</td>
<td>Saves the SP CEM state to memory starting at address $Rs + imm$, sets the CEM Status to “CEM Saved”, and clears the CEM interrupt registers. These steps are performed in one atomic operation — the hardware cannot be interrupted until the operation is completed. The hypervisor should choose a memory location that is not accessible to any other software or privilege levels. The saved SP CEM state is composed of the CEM Status, CEM Interrupt Address (along with Process ID and Virtual Machine ID bits), and the CEM Interrupt Hash. These are first combined as a block and encrypted with the device key. A keyed hash using the device key is also created and appended to the block. <em>Available in hypervisor privilege level only, and CEM Status of “Normal” or “Interrupted CEM”</em></td>
</tr>
<tr>
<td>Restore SPRegs Rs,imm</td>
<td>Loads the SP CEM state from memory at address $Rs + imm$. The keyed hash is verified before decrypting the contents and loading them into the on-chip CEM registers. To create SP state for a new virtual machine, a block of all zeroes should be “restored”. The hardware will detect this and accept it, setting the CEM status to “Normal” for the new block with the CEM interrupt registers cleared. <em>Available in hypervisor privilege level only, and CEM Status of “CEM Saved”</em>.</td>
</tr>
</tbody>
</table>

been restored or initialized. The processor can remain in this state while the hypervisor performs operations to change context or when the hypervisor does not want a virtual machine to be able to use CEM at all. None of the normal SP instructions are permitted while in “CEM Saved” state, except for the restore operation by the hypervisor.3 If executed, they will raise an exception.

When switching virtual machines, the TML or any separation kernel hypervisor must prevent all unauthorized sharing of data. This means that it must flush the cache before switching, unless the on-chip caches support tagging with a virtual machine ID. The hypervisor must then avoid timing channels by ensuring that the time for a partition switch is the same regardless of CEM activity. For example, flushing the cache when there are secure data lines may take longer since modified lines must

---

3It is unspecified whether or not initialization instructions for the DMK/DRK or the UMK are permitted while in the “CEM Saved” state, as these are typically only available when invoked using physical access to the device or during bootup. This would be implementation specific.
be encrypted and written back to memory by SP hardware — a process that takes longer than writing back non-CEM cache lines. The hypervisor should compensate by inserting an appropriate delay when there is less CEM activity.

Additional details regarding integrating the user-mode SP architecture with the SecureCore software stack are available from Bhaskara et al. [22].

### 7.2.2 Virtualization of Authority-mode SP

The virtualization of CEM is only one component necessary to virtualize the authority-mode SP architecture. For authority-mode, the remote-trust and transient-trust guarantees for the authority must be maintained even when a hypervisor is providing multiple virtual machines. While the hypervisor is trusted by the local user and by remote parties to assist with display of data, it is not considered trusted for maintaining authority-mode’s binding of the authority’s secrets to policies and to the device (via the on-chip DRK). Therefore, the hypervisor must not have control over, or access to, DRK-derived keys and must not be able to manipulate the DRK register.

For transient trust, the authority is guaranteed that the SRH value on-chip cannot be modified or replayed by any non-TSM software. Thus the hypervisor cannot be permitted to save SP state that includes the SRH to any off-chip storage where it could be subject to replay attacks, and it cannot have any ability to observe or modify
the SRH value. Only a TSM may access the SRH, and even it must only use the value in on-chip storage rather than on disk or in memory to prevent replay attacks. These requirements are in conflict with the need for the hypervisor to provide each virtual machine with a unique abstraction of the processor. Even if all TSMs could be trusted to share access to the on-chip SRH, in apparent violation of strict separation\(^4\), no single TSM in a virtual machine would be able to update the SRH, as it is the root of the secure local storage of all virtual machines on the device. The storage tree of the other machines would be inaccessible to any one TSM, and thus it could not produce a valid root hash over all of the trees.

Similarly, the hypervisor cannot be permitted to save or restore the data inside the CEM Buffer register when it calls `Save_SPRegs` or `Restore_SPRegs`. The CEM Buffer contains intermediate data — typically DRK-derived keys, nonces, or SRH values — which may be critical to operations and must not be subject to replay attacks. If the CEM Buffer were saved by the hypervisor, attacks on it would be effectively equivalent to attacks on the SRH register if it were saved, since an interrupt could be injected into the TSM execution just before writing the SRH value or just after reading it, when this data is still sitting in the CEM Buffer. In fact, the TSM may also use on-chip registers for this purpose, holding derived keys and hash values, so they too must be protected from replay attacks when virtualizing authority-mode SP.

To provide an SRH to each virtual machine’s TSM, we create two types of TSM code — application TSM code that runs in virtual machines, and master TSM code that runs in the hypervisor.\(^5\) The master TSM is in charge of updating the on-chip SRH register and providing each virtual machine with a virtual SRH register. The `SRH_Set` and `SRH_Get` instructions are virtualized for the application TSMs.

An application TSM runs normally in CEM, with access to all of the authority-mode SP instructions. When it executes an `SRH_Set` or `SRH_Get` instruction, the processor suspends CEM and raises an “SP SRH Virtualization” exception. The exception is captured by the hypervisor, which then launches the master TSM. Since there is already a suspended CEM thread in the SP interrupt registers, the master TSM runs with interrupts disabled so that it cannot generate a conflicting interrupt state. A special mode bit is used on-chip to keep track of whether or not a master TSM is executing.\(^6\)

---

\(^4\)Sharing of the SRH between TSMs in multiple virtual machines would create a storage covert channel and therefore must be prevented. The DRK is treated the same way as the DMK in Section 7.2.1, and is therefore not a concern as a storage channel. First, it is protected by the DRK Lock instruction, such that it cannot be written by software in the virtual machines after the system has booted. Second, overwriting the DRK provides 1-bit of information across virtual machines — that previously valid TSM code is no longer valid — passing one bit of information total, rather than a covert channel with capacity of 1-bit at a time.

\(^5\)Here, application-TSM and master-TSM code are distinct types of code, signed differently for the hardware’s code integrity checking. Kernel-TSM code, as described in Section 6.4.1, is not specially signed and could be of either hardware type, depending on whether or not it should access the virtualized SRH (and possibly trigger interrupts) or the hardware SRH (and run with interrupts disabled).

\(^6\)The additional mode bit for the master TSM can be either “Master TSM Active” (1) or “Inactive” (0). A master TSM can only be started when the regular CEM Status is either “Interrupted CEM”, “Normal”, or “CEM Saved”; it cannot be used during “Active CEM”. 

138
Master and application TSM code is distinguished by a new parameter to the Begin.CEM instruction indicating the type of TSM, as well as the addition of this parameter to the hash input for generating the signature on each cache line of TSM code. The signature will be verified by the SP hardware during CIC checking, to ensure that each TSM executes in the correct mode only.

When an application TSM calls SRH.Set, it first writes the desired new SRH value to the CEM Buffer register. It then executes the instruction and is suspended as described above. The hypervisor calls the master TSM, which has access to the CEM Buffer register to read the value provided by the application TSM. The master TSM next updates its secure local storage to reflect the new SRH value for the virtual machine (see Figure 7.2), and updates the on-chip SRH accordingly. Finally, the master TSM exits and the hypervisor can return to the virtual machine, which can resume the suspended CEM thread for the application TSM. Similarly for a call to SRH.Get, the application TSM is immediately suspended. The master TSM verifies the integrity of its secure storage against the on-chip SRH, retrieves the virtual machine’s SRH value from its storage, and then writes the value to the CEM Buffer before returning.

The hypervisor must specify a virtual machine ID for each call to the master TSM so the SRH value can be associated with the correct virtual machine when set and can be retrieved properly later. If the hypervisor provides an incorrect virtual machine ID, it would cause the master TSM to write the wrong machine’s SRH back to the processor, which would pass information between virtual machines as a covert channel; this is not a concern since the hypervisor is trusted to provide separation. If it did violate this trust, this mismatch would only enable the hypervisor to swap the secure storage assigned to different TSMs, which are all already trusted to enforce the attached policies, regardless of which virtual machine they are executing in. The individual application TSMs can also tag their data to verify that the secure local storage it is accessing is a match for the virtual machine in which the code is supposed to be executing.

To protect the CEM Buffer and the interrupt hash of the TSM’s general registers from replay attacks, we permit only the master TSM to call Save.SPRegs and Restore.SPRegs. These instructions now also encrypt and save the CEM Buffer register. When the hypervisor requests that the master TSM save SP state for a virtual machine, the master TSM will save a hash of the SP state in its secure local storage and update the on-chip SRH. It verifies this hash when asked to restore the state for a virtual machine. In this way, it can ensure that only the latest saved state is valid to be restored. The master TSM also permits the hypervisor to request that the SP state be reset, clearing all but the SRH from its secure local storage for that machine and setting the on-chip CEM Buffer, CEM Status, and interrupt registers all to zero. This is used when starting a new virtual machine or when a virtual machine should be rebooted and have its volatile data discarded, as would be the case with a hard reset of a physical processor.7

---

7By storing the hash of a virtual machine’s SP state in secure local storage, the master TSM permits a hypervisor to suspend and resume a running virtual machine, even if the physical processor
The API for the master TSM is shown in Table 7.2. It is intended to be used only from inside the hypervisor, since the data for its secure local storage must be accessible when called on behalf of any virtual machine. Additionally, it uses the new `Save_SPRegs` and `Restore_SPRegs` instructions, which are only available in the hypervisor’s privilege level.

Table 7.2: Interface for the Hypervisor’s Master TSM for Authority-mode SP Virtualization.

<table>
<thead>
<tr>
<th>Master TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterTSMLib_SRH_Get (int VM_ID)</td>
<td>Fetch the stored SRH value for virtual machine VM_ID from Master TSM’s secure local storage, first verifying the secure storage structure against the on-chip SRH value, and then write the virtual SRH value to the CEM Buffer register.</td>
</tr>
<tr>
<td>MasterTSMLib_SRH_Set (int VM_ID)</td>
<td>Store the SRH value for virtual machine VM_ID from the CEM Buffer register to the Master TSM’s secure local storage, updating the on-chip SRH value accordingly.</td>
</tr>
<tr>
<td>MasterTSMLib_Save_SP_State (int VM_ID, SP_State *buf)</td>
<td>Saves the SP CEM state for virtual machine VM_ID to memory at location buf and records a hash of the saved state to secure local storage, updating the on-chip SRH value accordingly. This function uses the <code>Save_SPRegs</code> instruction which writes the state to memory, sets the CEM Status to “CEM Saved”, and clears the CEM interrupt registers and the CEM Buffer register. The saved SP CEM state for authority-mode SP is composed of the CEM Status, CEM Buffer, CEM Interrupt Address (along with Process ID and Virtual Machine ID bits), and the CEM Interrupt Hash. These are first combined as a block and encrypted with the DRK. A keyed hash using the DRK is also created and appended to the block. The <code>Save_SPRegs</code> instruction is only available to the master TSM for authority-mode SP.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 7.2: (continued)

<table>
<thead>
<tr>
<th>Master TSM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MasterTSMLib_Restore_SP_State</strong> (int VM_ID, SP_State *buf)</td>
<td>Restores the SP CEM state for virtual machine VM_ID from memory at location buf, first verifying the hash of the saved state in secure local storage. This function uses the <em>Restore_SPRegs</em> instruction, which loads, verifies, and decrypts the state from memory. The <em>Restore_SPRegs</em> instruction is only available to the master TSM for authority-mode SP.</td>
</tr>
<tr>
<td><strong>MasterTSMLib_Reset_SP_State</strong> (int VM_ID)</td>
<td>Deletes the saved SP CEM state for virtual machine VM_ID from the secure local storage (if present) and clears the on-chip SP CEM state (setting CEM Status to “Normal” and clearing the CEM Buffer and interrupt registers). This function uses the <em>Restore_SPRegs</em> instruction with zeroed data to clear the on-chip registers. The saved value of the SRH for this virtual machine is not affected.</td>
</tr>
<tr>
<td><strong>MasterTSMLib_Delete_VM</strong> (int VM_ID)</td>
<td>Deletes the saved SP CEM state (if present) as well as the saved SRH for virtual machine VM_ID from the secure local storage. The on-chip SRH value is updated accordingly, but the on-chip SP CEM state is not affected. This is used to clear space and possibly reuse a VM_ID when an entire virtual machine is deleted on the platform and will never be needed again.</td>
</tr>
</tbody>
</table>

To implement these functions, the master TSM maintains its own local secure storage to keep persistent data about each virtual machine’s SP state and SRH value. As shown in Figure 7.2, the master TSM stores a list of structures, each containing a virtual machine ID, saved SRH value, and the hash of the saved SP CEM state. It computes a hash over this entire list and saves that root hash in the on-chip SRH register. Any time a call is made to the master TSM, it first verifies the list of structures against the on-chip SRH value. Then if a change is made, it updates the structure, recomputes the hash over the entire list, and updates the on-chip SRH. It is important that the on-chip value is checked each time a Get or Restore command is issued so that the output to the application TSM is still rooted in on-chip storage and prevents all attempted replay attacks on the secure storage within each virtual machine.
Figure 7.2: Secure Storage for the Master TSM
7.3 A Combined User-mode and Authority-mode SP Device

There can be some usage scenarios where it is desirable to have the use of both user-mode SP and authority-mode SP on the same device. In this case the two modes would operate mostly independently and there would be separate authority- and user-TSMs. This allows local users of such a device to securely access authority-owned secrets (e.g., for business purposes) and their own personal secrets (e.g., for personal credentials) at different times without owning multiple devices.

Much of the SP hardware can be shared that implements CEM, with additional registers and tags to distinguish between the two CEM security domains. We add the hardware required to support simultaneous suspended CEM threads and separate memory protection, but only allow one CEM thread to be active at any one time in either user-mode or authority-mode.\(^8\) It is permitted to have one active CEM thread in either mode while another CEM thread is suspended in the other mode.

The hardware architecture for the dual-mode SP device is shown in Figure 7.3. The CEM Mode/Status register has extra bits to store the state of user-mode and authority-mode independently, with certain combinations of states invalid or reserved (e.g., “Active User-mode” with “Active Authority-mode” is invalid). The interrupt registers have been duplicated, and the volatile and non-volatile registers for the master secrets of each mode are provided on-chip. An additional bit is added to each of the cache tags in the L1 and L2 caches to distinguish secure lines for the two modes. The Encryption & Hashing Engine will also distinguish between the modes to

\(^8\)There may be both a user-mode application and authority-mode application running on the device simultaneously, but only one of the two can be scheduled on the processor and have its TSM executing in CEM at any given time. Therefore, even with additional hardware support, it would not make sense to simultaneously support an active user-mode CEM thread and an active authority-mode CEM thread on a single-core device. One TSM must exit or be suspended before another TSM can be started or resumed. Since user-TSMs and authority-TSMs are trusted by different entities, no single TSM should operate with the privileges and secrets of both trust domains. Furthermore, any line of TSM code can be signed as either a user-TSM or an authority-TSM (or not signed at all for normal code).

143
use the appropriate device key, either the DMK or the DRK. This device key is used for secure memory operations, secure instruction verification in CIC, and interrupt protection of general registers.

The Begin\textsubscript{CEM} instruction is modified to indicate which mode should be entered using a sub-op parameter. TSM code will start with either a Begin\textsubscript{CEM.user} or Begin\textsubscript{CEM.auth} instruction, causing the SP hardware to enter CEM using the correct mode and to use the correct device key for CIC checking of subsequent instructions. If the incorrect Begin\textsubscript{CEM} instruction is used, the CIC checking of the next instruction will fail.

Most other instructions remain unchanged, either serving the same purpose in both modes (e.g., End\textsubscript{CEM}, Secure\textsubscript{Load/Store}) or being accessible only in the appropriate mode (e.g., DRK\textsubscript{Derive}, SRH\textsubscript{Get}, UMK\textsubscript{Get}).

7.3.1 Protecting the User’s UMK on a Dual-mode SP Device

We offer an alternative use for a combined device, with an addition to the hardware, to solve a potential security and usability problem with user-mode SP. Owners of a user-mode SP device want to be sure that their device remains secure after they initializes it with a DMK and signs their user-TSM. If adversaries obtain temporary possession of the device, they could re-initialize the DMK, sign a new user-TSM, and return the device to its owner. The users would then enter their passphrase to generate their UMK, not knowing that the device was compromised. Rather than using their own user-TSM, the user would inadvertently access the malicious TSM installed by an adversary, which may reveal their keys or use them surreptitiously.

What is needed is a mechanism to verify that the TSM is unchanged before giving it access to a user’s UMK, which sits at the root of the keychain. Since an authority-TSM is trusted and verified remotely, it can serve this purpose by verifying the state of the user-mode SP secrets on a dual-mode device.\footnote{This scenario, where an authority-TSM verifies the user-mode SP secrets, will usually be distinct from other uses of authority-mode SP. A different organization would be setup as the authority to fill this role, one that the users trust and selects personally. This authority would initialize the device at its depot or in coordination with the users, and would provide an authority-TSM for remotely verifying the state of the device. If the device is used to access authority-owned secrets, the users may not trust that same authority to have any role in protecting their own secrets, and therefore may choose not to use this feature.}

With this change, when a user-TSM is started, it will initially have no access to the UMK until the device state can be verified. Without access to the UMK, a user-mode TSM can execute but has no special access, other than the ability to use CEM to protect its data from other software on the device; it cannot access any of the user’s secrets. Access to the UMK will only be provided to user-TSMs on the device after the authority-TSM has provided authorization.

The determination is ultimately whether or not users trusts a particular TSM. A proxy for this trust is whether or not the same DMK value is installed in the device as when the users initialized the device and signed that TSM. A pairing relationship can therefore be formed between the DMK on the device that signed a user-TSM and...
one or more UMK values corresponding to user keychains that should be accessible to that TSM.

We represent this trust relationship by a token, cryptographically signed to indicate that a DMK-UMK pair is authorized by the user. If this Verification Token exists for a given pair, then the TSM signed with the DMK should be permitted access to the given UMK. The authority-TSM will be the only software entity able to generate a valid verification token.

During operation, users will enter their passphrase into the device, via the secure interface. This will be used to regenerate their UMK, which resides in the on-chip register. Then the user-TSM is launched, executing in CEM, but without access to the UMK register. A verification token, if it exists, can then be provided to the TSM. The TSM supplies this token to the hardware, which then releases the UMK value to the TSM.

The decision to trust a particular TSM with a user’s UMK need only be made once, and can then be made permanent by creating a verification token for the UMK-DMK pair. If the token is cryptographically verifiable and unforgeable, it can be generated in advance by the authority-TSM and then released to untrusted storage to be used during operation without further invocation of the authority-TSM.

When the user-TSM is operating and wants to access the UMK register, it calls the UMK_Get instruction, providing the verification token as a parameter. The hardware regenerates the token using the current DMK and UMK registers and verifies that the one provided is correct before writing the UMK to general registers for the user-TSM. If the token does not match, the hardware will write zeroes as the result instead.

In order to securely generate these tokens, a few new requirements are placed on the authority-mode TSMs. First, the authority-TSM must be able to generate a verification token from the UMK currently entered into the device, but only upon request of the user. Second, it must be able to verify the state of the DMK during operation, to determine if it is legitimate before generating a token. Third, it must be aware when the user-mode DMK is initialized legitimately, as a basis for the verification.

For token generation, we add a new instruction, Generate_UMK_Token, for authority-TSMs that causes the hardware to generate a MAC over the UMK, keyed with the DMK. The authority-TSM can then write the token to storage or unprotected memory to release it for use by the user-TSM. This instruction is available only to authority-TSMs and only if users indicate that they want to initialize their UMK on this device as they enter their passphrase. Users should use the secure I/O mechanism for this purpose, such as a special physical button, distinguishing their normal use from initialization.

We add another new instruction, DMK_Derive, for authority-TSMs to verify the state of the DMK during device initialization and token generation. This instruction functions the same as DRK_Derive; it takes a nonce as a parameter, which is used to generate a MAC using the DMK, and writes the result to the CEM Buffer register. The authority-TSM uses this instruction to obtain a token that represents the DMK without having direct access to the DMK value, which would break the security guarantees of user-mode SP.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin_CEM.user</td>
<td>Begins user-mode CEM with the next instruction, using the DMK for Code Integrity Checking and other CEM operations. The user CEM status must be “Normal” and will be set to “Active”. The authority CEM status must not be “Active”.</td>
</tr>
<tr>
<td>Begin_CEM.auth</td>
<td>Begins authority-mode CEM with the next instruction, using the DRK for Code Integrity Checking and other CEM operations. The authority CEM status must be “Normal” and will be set to “Active”. The user CEM status must not be “Active”.</td>
</tr>
<tr>
<td>Generate_UMK_Token Rd</td>
<td>Creates a Verification Token for the current DMK and UMK values by generating a MAC over the UMK, keyed with the DMK. The resulting token is written to a consecutive pair of registers, starting in Rd. Available in CEM Status of “Active Authority-mode” only, and only if the user has enabled initialization when the UMK was entered.</td>
</tr>
<tr>
<td>UMK_Get.sel Rd, Rs1, Rs2</td>
<td>Verifies the Verification Token provided in Rs1 and Rs2 against the current DMK and UMK values by generating a MAC over the UMK, keyed with the DMK. If the provided token matches, the current UMK is written to Rd (either the high-order 64-bits or low-order 64-bits, as determined by sel), otherwise Rd is zeroed. Available in CEM Status of “Active User-mode” only.</td>
</tr>
<tr>
<td>DMK_Derive Rs1, Rs2</td>
<td>Derives a new key from the DMK by computing a keyed hash over the contents of Rs1 and Rs2. The result is stored in the lower 128-bits of the CEM Buffer register. Available in CEM Status of “Active Authority-mode” only.</td>
</tr>
</tbody>
</table>
Initialization and token generation occur in two steps. In the first step, the authority-TSM identifies the trusted user-TSM by its DRK. During device initialization for user-mode SP, when users creates a new DMK and signs their TSMs, they will also invoke the authority-TSM (which will be previously initialized at the authority’s depot). The authority-TSM will generate a DMK-derived key using a known nonce and store the key in its secure local storage. The owners must authenticate themselves to the authority-TSM, or to the authority itself offline, to indicate that they are legitimately initializing their device, rather than an adversary maliciously doing so. According to the initialization process of user-mode SP, these steps should take place in a secure environment using only trusted software that will not leak the DMK or sign additional or incorrect user-TSM code. No attacks are possible in this environment, so the authority-TSM can be assured that the DMK is correct at this time, that only this DMK should be trusted, and that the DMK will be associated only with a trusted TSM.

The second step is for users to authorize the use of their keychains on the device using the previously trusted TSM. After the device is put into normal use, possibly in hostile environments, users can approach the machine to make use of their personal keychains. The users contact the authority to request that the device’s state be remotely verified. The authority uses remote mutual authentication, as described in Chapter 3, to check that the DRK is still correct and that the authority-TSM is trusted. The result of this check must be communicated securely to the users, since the users should only proceed if the authority-TSM is trusted to accurately verify the user-mode state of the device. If the users trusts the authority, the result of its authentication, and that the owner of the device installed a trustworthy TSM, they enter their UMK into the device (using the secure I/O to permit token generation) and request that the verification token be generated. The authority-TSM then uses $DMK_{Derive}$ with the same nonce as earlier to verify that the DMK is unchanged. If it is, then it generates and releases the verification token. If the authority-TSM finds that the DMK has changed, it will not generate a token.

The concept of using verification tokens to authenticate a user-mode SP device before providing access to the UMK can also be applied on a device with user-mode SP only. Instead of providing Generate_UMK_Token to the authority-TSM, it is provided to user-TSM when the secure I/O initialization mechanism is invoked. Since there is no authority to remotely verify the state of the DMK, the users should only generate a token when they first initialize their device in the secure environment. This puts an additional limitation that all users must be present during the device initialization process to generate the verification tokens for all UMKs that will be used on the device. Generating additional tokens afterwards would not be safe since the DMK cannot be verified once the device has been taken outside of the secure environment.

The users can be informed of the result of the remote verification of the authority-mode state securely through another device that they have already verified with the authority. To bootstrap the process, their first device may need to be initialized in a secure physical location that provides a secure and verified link with the authority. Alternatively, the authority could send its result through the device itself (e.g., by revealing a secret phrase known only to the authority and the user) or via an offline mechanism (e.g., telephone, text message) that has different failure modes than the device itself, such that an adversary that can compromise the device being initialized is not also likely to interfere with this alternative communication channel.)
From this point forward, the users can use the user-TSM on the device normally. They can enter their UMK at any time (without permitting token generation), which will only be accessible to the user-TSM if a valid verification token already exists. Since the authority-TSM cannot generate a new token without the secure I/O initialization mechanism being invoked, the users do not need to verify the state of the authority-TSM on each use.

If the users make use of their UMK and keychains on multiple devices, they can generate a verification token for each of those devices, keeping the set of all such tokens alongside their keychains in permanent storage. Any device that they have previously authorized with a token will be able to use their keychains on their behalf, whereas any untrusted device or any compromised device will not have access.

7.4 Security Analysis

7.4.1 Virtualization of SP

For virtualization, a hypervisor is permitted to save and restore the SP state of a suspended CEM thread when it is switching between virtual machine contexts. It is critical that neither the partially-trusted hypervisor nor any other untrusted software be able to manipulate the SP state or maliciously affect the execution of a TSM.

When the state is saved, the hardware first encrypts and MACs the state data using the on-chip device key. This protects the confidentiality and integrity of the data, ensuring that the state cannot be spoofed or modified without detection when restored. Thus the hypervisor can only artificially clear a suspended CEM thread, but cannot insert one or change the program counter address or any general register contents of a suspended TSM. Clearing a CEM thread is merely a denial of service attack, which the hardware does not aim to protect.

Since the SP state cannot be modified, an adversary can only inject an SP state that was created by the hardware at some time in the past, possibly replaying old data. For user-mode SP, replaying an old SP state presents only a small threat; the replayed state would include the general registers together with the program counter, thus restarting the TSM where it had left off previously. Further effects of the execution past the point it was restored are effectively discarded, aside from any changes made to data in memory or data stored on disk. If any of this other data is in a state that is inconsistent with the replayed CEM state, it could affect the future decisions of the TSM code. Given that the hypervisor is trusted software, albeit with a different threat model than the TSM, we do not view this as a serious threat.

For authority-mode SP, replaying or rolling back the SP state would be a violation of transient trust guarantees. Data and keys that were previously revoked might be reinstated, and usage counters, audit data, and other access control mechanisms could be maliciously reverted. For this reason, only the master TSM is permitted to save and restore the SP state, and it will use its secure local storage to prevent replay. This storage itself is safe from replay attacks because it is rooted in a hash stored in the on-chip SRH register.
The replay protection of the SP state covers the general registers and the CEM buffer, any of which may contain SRH values, derived keys, or access control data being processed by the TSM. Virtualization of the \texttt{SRH\_Get} and \texttt{SRH\_Set} instructions must similarly be replay-resistant to maintain the same security properties for the virtualized SRH register that each virtual machine would have if it used a real, non-virtualized on-chip SRH register. As described previously, each time one of these virtualized instructions is invoked, the master TSM must recheck its stored values against the on-chip master SRH. Since the master TSM runs with interrupts disabled, there will be no opportunity for untrusted software to disrupt this check once it has started execution of the master TSM. The untrusted software also cannot disrupt the transmission of the parameters or results of these instructions, as they take place using protected SP registers (e.g., the CEM Buffer), which are only accessible to TSM code.

7.4.2 Protecting the User’s UMK on a Dual-mode SP Device

The goal of the combined device with UMK protection is to ensure that users never provide access to their UMK to a TSM that they do not trust, even if their device is stolen and then returned to them with a different Device Master Key. With our new design, users first enter their passphrase into the secure I/O hardware to generate their UMK on-chip. The UMK register will not be revealed to any software without a valid verification token, and is safe in the hardware until a token is entered.

During operation, only TSM code signed with the current DMK can enter user-mode CEM to use the \texttt{UMK\_Get} instruction that presents a verification token to the hardware to attempt to read the UMK. This TSM cannot be modified without changing the DMK, assuming no new code is signed. When \texttt{UMK\_Get} is executed, the hardware verifies the token against the current UMK and DMK. If the token verifies properly, this indicates that the current DMK is identical to one that the users had previously decided to trust with this UMK; this indicates that the TSM is also the same and is trusted, since each DMK will only be used to sign a single TSM.

Verification tokens cannot be forged since they are cryptographically signed with the DMK, and the trusted DMK is only known outside of the SP hardware during initialization. The token generation process uses a cryptographic one-way function to ensure that neither the DMK nor UMK can be obtained from an existing token. Therefore the tokens also do not need to be kept secret. An adversary cannot use the token to obtain any secrets directly. Furthermore, the token is only useful in a device with the matching DMK, and then only when the UMK is entered — an adversary cannot enter the passphrase to generate this UMK without the user’s help.

Having stolen the device, adversaries who installs their own DMK (that signs a malicious TSM) cannot generate a valid verification token to go along with their new DMK and a user’s UMK since they would need to already know the UMK in order to generate that token. If the adversaries already knew the UMK, there would be no need to launch the attack. Therefore if the adversaries return the device to the legitimate user with the bad DMK, the user’s real verification token will not match, nor will any other existing token, and the UMK will not be released to the malicious
TSM. Users will clear their UMK after each session, before relinquishing physical control of the device, to ensure that a device is not stolen with the UMK already entered. Additionally, executing either the DMK\_Set or DRK\_Set instruction will cause the UMK register to be cleared.

During initialization and token generation, additional attacks must be considered. Users must start the user-mode initialization process having the authority-mode CEM already initialized with a valid DRK and authority-TSM according to the procedure in Chapter 3. The users authenticate themselves to the authority’s TSM to demonstrate that the legitimate owner of the device is physically present on the device at the time of initialization. Users should only do this in a secure physical environment, as described in Section 7.3.1. The users can then set the DMK, and signal the authority-TSM that they have done so. The authority-TSM uses DMK\_Derive at this time to identify the current DMK. From this point forward, the authority will remember that the users trust this DMK, although it does not know the value of the DMK itself. An adversary cannot authenticate to the authority, and cannot affect the operation of the device during initialization in the secure environment, and therefore cannot affect which DMK the authority trusts.\(^\text{12}\) The users can use the DMK to sign their TSM and then reset the device, such that the DMK will then only be known in SP hardware. The adversaries therefore cannot obtain the DMK from the initialization process or from the device afterwards.

For token generation, the users now asks the authority to authenticate the state of the authority-TSM on device. If it properly authenticates, the authority must inform the users securely of this fact. This tells the users that the authority-TSM that they are about to trust has not been compromised. Once they enters their UMK and initializes token generation through the secure I/O process, the authority-TSM will be able to generate a token at its discretion, and the users must trust it to do so only after verifying the DMK. The authority-TSM uses the DMK\_Derive instruction again to verify that in fact the DMK is still unchanged from one of the DMK values that the user had previously indicated as trusted. If not, the authority-TSM is programmed not to call the \texttt{Generate\_UMK\_Token} instruction. Once this instruction is used on a device with the proper DMK, the token can be released arbitrarily without concern of attacks, as previously described.

If the DMK is changed between the initialization and the token generation steps, the authority-TSM will detect this before requesting a new token from the hardware. If the DRK is changed in that time, the authority will be unable to authenticate the authority-TSM and will tell the users not to enter their UMK. The authority-TSM is not used during regular user-mode operation, aside from token generation, so later attacks are not a concern for this process. A modified authority-TSM cannot generate tokens unless the secure I/O mechanism is invoked to do so, and the legitimate users will not do so when their UMK is entered unless they have authenticated the authority-TSM first. A modified DMK and user-TSM will cause the verification token to be rejected, as previously described, since the token is re-checked on each use.

\(^{12}\)If the process is somehow faked or interrupted, the real authority will simply not trust this DMK, making it refuse to generate tokens in the future steps.
7.5 Overview of SP Enhancements

To summarize the enhancements to the SP architecture, we provide a full list of SP instructions and SP state combining user-mode SP [102], authority-mode SP (Chapter 3), Secure Areas (Section 5.4.3), SP virtualization (Section 7.2), and dual-mode devices (Section 7.3).

If Secure Areas are implemented, the new Secure Area instructions will replace Secure_Load and Secure_Store. In a combined user-mode and authority-mode device, Secure Area regions are separate for the two modes, requiring separate on-chip storage for the Secure Area meta-data for each mode and a separate Saved Stack Pointer for each mode (see Figure 5.6).

Table 7.4: Summary of All SP Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instructions to Enter CEM and for Initialization</strong></td>
<td></td>
</tr>
<tr>
<td>Begin_CEM.user</td>
<td>Begins user-mode CEM.</td>
</tr>
<tr>
<td>Begin_CEM.auth type</td>
<td>Begins authority-mode CEM. Enters either the master-TSM or application-TSM mode, as specified by type.</td>
</tr>
<tr>
<td>DRK_Set Rs1,Rs2,sel</td>
<td>Set the selected double-word region, sel, of the Device Root Key register from general registers Rs1 and Rs2.</td>
</tr>
<tr>
<td>DRK_Lock</td>
<td>Locks the DRK register.</td>
</tr>
<tr>
<td>DMK_Set Rs1,Rs2,sel</td>
<td>Set the selected double-word region, sel, of the Device Master Key register from general registers Rs1 and Rs2.</td>
</tr>
<tr>
<td>DMK_Lock</td>
<td>Locks the DMK register.</td>
</tr>
<tr>
<td><strong>Instructions Available in Active CEM (User-mode or Authority-mode)</strong></td>
<td></td>
</tr>
<tr>
<td>End_CEM</td>
<td>Exits CEM.</td>
</tr>
<tr>
<td>Secure_Load Rd,Rs,imm</td>
<td>Secure load from memory, at address Rs + imm, to a general register, Rd. Optionally replaced by Secure Areas.</td>
</tr>
<tr>
<td>Secure_Store Rd,Rs,imm</td>
<td>Secure store to memory, at address Rs + imm, from a general register, Rd. Optionally replaced by Secure Areas.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 7.4: (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instructions Available in Active CEM for Secure Areas</strong></td>
<td></td>
</tr>
<tr>
<td>SecureArea_Add Rs1,Rs2, region</td>
<td>Initialize the specified Secure Area (for <code>region</code>), specifying the start address in <code>Rs1</code> and the size in <code>Rs2</code>.</td>
</tr>
<tr>
<td>SecureArea_Rereolate Rs1, <code>region</code></td>
<td>Change the starting address of the specified Secure Area region to the value in <code>Rs1</code>. The size remains unchanged.</td>
</tr>
<tr>
<td>SecureArea_Remove <code>region</code></td>
<td>Disables and clears the specified Secure Area region.</td>
</tr>
<tr>
<td>SecureArea_CheckAddr Rd, <code>region</code></td>
<td>Retrieves the starting address of the specified Secure Area region.</td>
</tr>
<tr>
<td>SecureArea_CheckSize Rd, <code>region</code></td>
<td>Retrieves the size of the specified Secure Area region.</td>
</tr>
<tr>
<td>Set_Secure_SackPtr Rs</td>
<td>Saves the current stack pointer to the new Saved Stack Pointer register and sets the stack pointer to the value in <code>Rs</code>. (see [48].)</td>
</tr>
<tr>
<td>Restore_SackPtr</td>
<td>Restores the original stack pointer from the new Saved Stack Pointer register, if set, and then clears the register. (see [48].)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Instructions Available in Active User-mode CEM Only</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UMK_Get.sel Rd, Rs1, Rs2</td>
<td>Verifies the Verification Token provided in <code>Rs1</code> and <code>Rs2</code>, writing the selected, <code>sel</code>, portion of the UMK to <code>Rd</code>.</td>
</tr>
</tbody>
</table>

(Continued on next page)
### Table 7.4: (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Available in Active Authority-mode CEM Only</td>
<td></td>
</tr>
<tr>
<td><strong>DRK_Derive Rs1,Rs2</strong></td>
<td>Derives a new key from the DRK and the contents Rs1 and Rs2, writing the result to the CEM Buffer register.</td>
</tr>
<tr>
<td><strong>SRH_Get</strong></td>
<td>Copies the SRH register into the CEM Buffer register. <em>Raises “SP SRH Virtualization” exception if not in master-TSM mode.</em></td>
</tr>
<tr>
<td><strong>SRH_Set</strong></td>
<td>Atomically copies the CEM Buffer register into the SRH register. <em>Raises “SP SRH Virtualization” exception if not in master-TSM mode.</em></td>
</tr>
<tr>
<td><strong>GR_Get Rs1,Rs2,sel</strong></td>
<td>Retrieves two words from the general registers into the selected double-word region, <em>sel</em>, of the CEM Buffer register.</td>
</tr>
<tr>
<td><strong>GR_Set Rd,sel</strong></td>
<td>Sets a general register with the selected word, <em>sel</em>, of the CEM Buffer register.</td>
</tr>
<tr>
<td><strong>Generate_UMK_Token Rd</strong></td>
<td>Creates a Verification Token for the current DMK and UMK values, writing to a pair of general registers starting in <em>Rd</em>.</td>
</tr>
<tr>
<td><strong>DMK_Derive Rs1, Rs2</strong></td>
<td>Derives a new key from the DMK and the contents of Rs1 and Rs2, writing the result to the CEM Buffer register.</td>
</tr>
<tr>
<td>Instructions Available to the Hypervisor</td>
<td></td>
</tr>
<tr>
<td><strong>UMK_Set Rs1,Rs2,sel</strong></td>
<td>Set the selected double-word region, <em>sel</em>, of the User Master Key register from general registers Rs1 and Rs2. <em>Available if the hypervisor implements the secure I/O features of user-mode SP to generate the UMK from the user’s passphrase.</em></td>
</tr>
<tr>
<td><strong>Save_SPRegs.user Rs,imm</strong></td>
<td>Saves the SP User-CEM state to memory starting at address Rs + imm.</td>
</tr>
<tr>
<td><strong>Restore_SPRegs.user Rs,imm</strong></td>
<td>Loads the SP User-CEM state from memory at address Rs + imm.</td>
</tr>
<tr>
<td><strong>Save_SPRegs.auth Rs,imm</strong></td>
<td>Saves the SP Authority-CEM state to memory starting at address Rs + imm. <em>Available to the Master TSM only.</em></td>
</tr>
<tr>
<td><strong>Restore_SPRegs.auth Rs,imm</strong></td>
<td>Loads the SP Authority-CEM state from memory at address Rs+imm. <em>Available to the Master TSM only.</em></td>
</tr>
</tbody>
</table>
The CEM mode register must also be expanded to accommodate the various combinations of SP states. Table 7.5 lists all of the possible states for a combined user-mode and authority-mode device with virtualization.

For code integrity checking, each cache line of TSM code must be signed with either the DMK for user-mode TSMs or with the DRK for authority-mode TSMs. The signature will be computed over the code, the virtual address of the cache line, and a set of flags (as shown in Figure 3.4). For user-mode TSMs, the only flag is to indicate if the cache line is a valid entry point that is permitted to follow a Begin_CEM instruction. For authority-mode TSMs, we add an additional flag to indicate if the TSM is an application-TSM or a master-TSM; the TSM type that is specified when Begin_CEM is called must correspond to this flag.

The cache tags that are added to each of the on-chip caches (see Figure 5.6) indicate whether or not a line has been verified as secure. If not tagged as secure, all new tags will be set to zero. For secure instruction lines, the first tag (3 bits) indicates if the line was verified as secure (with one bit to indicate if the line is secure, one bit to indicate an entry point, and one bit to distinguish application- and master-TSMs). For secure data lines, the first tag (2 or more bits) indicates which Secure Area region the line is part of (where zero indicates a non-secure line and values greater than zero specify a region number). A second tag indicates if the line is secure for user-mode CEM or authority-mode CEM. A final tag is used to indicate if a line is secure as secure CIC-verified instructions or as secure memory data. This last tag is implicit, and therefore not needed, in split instruction and data caches. Additionally, caches that are tagged with physical addresses must include an additional tag for the offset of the line within the Secure Area region, as described in Section 5.4.3; this ensures that splicing attacks cannot occur between previously-verified secure cache lines.

To properly check these cache tags and to protect CEM state during an interrupt, a number of hardware operations are modified for the base SP architecture. These changes are summarized in Table 7.6 below and are explained in more detail in the SP Processor Architecture Reference Manual [52].

7.6 Summary

We provide new solutions to use SP on a virtualized device, allowing the hypervisor to virtualize the SP state and permit each virtual machine to make use of a user-mode TSM or an authority-mode TSM without violating the security guarantees of either SP architecture. This permits significantly more flexibility in the usage scenarios for an SP device that includes a hypervisor, such as the SecureCore architecture described in Chapter 6. By allowing TSMs in multiple virtual machines, each with their own secure storage and roots of trust, trusted software can operate at different security levels for MAC policy when placed in multiple virtual machines, and multiple emergency partitions are possible that might be enabled with access to different sets of secrets at different times.

Furthermore, we show how to integrate authority-mode SP and user-mode SP on a single device to protect a user’s User Master Key on an SP device where physical
Table 7.5: List of Possible SP States for the CEM Mode Register

<table>
<thead>
<tr>
<th>CEM Mode Bits</th>
<th>SP CEM State</th>
<th>User</th>
<th>Auth. App</th>
<th>Auth. Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 0</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>00 00 1</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>00 01 0</td>
<td>None</td>
<td>Active</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>00 01 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00 10 0</td>
<td>None</td>
<td>Suspended</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>00 10 1</td>
<td>None</td>
<td>Suspended</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>00 11 0</td>
<td>None</td>
<td>Saved</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>00 11 0</td>
<td>None</td>
<td>Saved</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>01 00 0</td>
<td>Active</td>
<td>None</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>01 00 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01 01 0</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01 01 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01 10 0</td>
<td>Active</td>
<td>Suspended</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>01 10 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>01 11 0</td>
<td>Active</td>
<td>Saved</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>01 11 0</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10 00 0</td>
<td>Suspended</td>
<td>None</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>10 00 1</td>
<td>Suspended</td>
<td>None</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>10 01 0</td>
<td>Suspended</td>
<td>Active</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>10 01 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10 10 0</td>
<td>Suspended</td>
<td>Suspended</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>10 10 1</td>
<td>Suspended</td>
<td>Suspended</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>10 11 0</td>
<td>Suspended</td>
<td>Saved</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>10 11 0</td>
<td>Suspended</td>
<td>Saved</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>11 00 0</td>
<td>Saved</td>
<td>None</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>11 00 1</td>
<td>Saved</td>
<td>None</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>11 01 0</td>
<td>Saved</td>
<td>Active</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>11 01 1</td>
<td>— Invalid</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11 10 0</td>
<td>Saved</td>
<td>Suspended</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>11 10 1</td>
<td>Saved</td>
<td>Suspended</td>
<td>None</td>
<td>Active</td>
</tr>
<tr>
<td>11 11 0</td>
<td>Saved</td>
<td>Saved</td>
<td>None</td>
<td>Inactive</td>
</tr>
<tr>
<td>11 11 0</td>
<td>Saved</td>
<td>Saved</td>
<td>None</td>
<td>Active</td>
</tr>
</tbody>
</table>
Table 7.6: Summary of Modified Processor Operations for SP

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifications to Support Secure Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction fetch</td>
<td>The next instruction to be executed is fetched into the processor. If in Active CEM mode, the “secure instruction” tag is required for the appropriate mode (user-mode or authority-mode) and type (entry point or not, for either an application or master TSM). In all other CEM modes, no “secure” tag is permitted. If a cache-line does not have the correct tags, it must first be evicted and then re-fetched.</td>
</tr>
<tr>
<td>Regular loads and stores</td>
<td>Loads and stores that access normal (non-secure) memory first check that the cache line has no “secure” tags set. If the “secure” tag is set, the line will first be evicted and then re-fetched without the “secure” tag. Thus previously secure data will load in encrypted state and instructions will be loaded without CIC verification.</td>
</tr>
<tr>
<td>Cache line eviction</td>
<td>When a cache line with the “secure data” tag set is evicted from the last level of on-chip cache, it will first be encrypted and a MAC calculated, both using the appropriate device key, before being written to off-chip memory. When a cache line with the “secure instruction” tag set is evicted, it does not need to be written back to memory and can simply be discarded. Secure instruction lines cannot be written or modified — instead they must be evicted and then re-fetched as data.</td>
</tr>
<tr>
<td>Modifications to Support CEM Interrupt Handling</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td>If an interrupt occurs during Active CEM, the CEM interrupt protection is triggered to secure the register state using the appropriate device key, encrypting the general registers, and saving the Interrupt Hash and Interrupt Address. The CEM mode is then changed to Suspended CEM before returning control to the system software.</td>
</tr>
<tr>
<td>Return from Interrupt</td>
<td>If in Suspended CEM when returning from an interrupt and the saved Interrupt Address for either authority-mode or user-mode matches the current return address, the processor will verify and restore the register state for the appropriate mode and resume Active CEM.</td>
</tr>
</tbody>
</table>
security is not assured after initialization. This feature addresses a prior limitation of user-mode SP, and allows users to have confidence that their keychains will not be made available to an adversary, by indirectly allowing the user to authenticate the device for each use.
Chapter 8

Conclusion

In this work, we have designed, implemented, and tested the authority-mode SP architecture as a means of remotely trusting portable devices to protect sensitive data. We use new security mechanisms, including master secrets stored on chip, to form a non-bypassable root of trust in hardware. The hardware in turn protects Trusted Software Modules, providing a secure environment where the master secrets can be used to access encrypted data. Data and keys are accessed under control of software but without exposing them to attack.

We leverage the flexibility of the trusted software to store complex data structures, enforce intricate and dynamic access control policies, interact with remote parties over the network, and provide APIs to less trusted software components. These functions would be too complicated to build into hardware directly; software allows complex interactions while giving the ability to upgrade protocols and cryptographic algorithms as needed across the design lifetime of the device. As a result of this flexibility, our security primitives are able to serve a wide range of usage scenarios for protecting remote access to data, since the software can be customized for each purpose.

A key feature of our architecture that enables new and interesting uses is the ability to bind together secrets and software to the hardware root of trust on the device. This feature is critical for remote trust, since the data owner has assurance that the data can only be used with the software it trusts. The owner knows that if the software is changed or moved to another device, there will be no access to the secrets. Neither the authority nor any other remote party needs to take action to enforce this binding or to cut off access to secrets. By giving the software access to derived keys, it can further extend the binding of hardware trust to its own locally-generated data with the same level of trust. We use this feature for secure storage, secure revocation of data, and secure mutual authentication with the authority.

During the design and implementation of the architecture, we have written and tested Trusted Software Modules for emergency management. This allows us to demonstrate many of the core features, including access control, data binding, remote authentication, and revocation of data. We anticipate that for real deployments, trusted software will perform more computation on data internally, such as data analysis and searches, within the trusted security domain. To protect the dissemination
of data, it is especially helpful that large quantities of data can be made available to
the trusted software in this fashion, with only smaller amounts of data being released
outside of the trust boundary to other software and the local user.

To further contain this data, other software security mechanisms can be built
around our Trusted Software Modules. Our work on the SecureCore project demon-
strates one such scenario where emergency data is further contained within a partition
of a separation kernel after it leaves the security domain of a Trusted Software Mod-
ule. In general, such software systems can rely on our hardware-protected trusted
software to store their root keys or data, and to perform their most security critical
operations. The rest of the software in the partition and in the kernel that sits out-
side the trusted domain then provides a framework to protect the less critical data;
it also controls access to the Trusted Software Module functions, provides a rich user
interface to the user, and supports the use of other software that does not need to be
protected.

As an example of this separation of duties in SecureCore, we use Trusted Software
Modules to protect sensitive data, to setup secure communication channels with the
authority and third parties, and to manage the emergency state. The other software
builds on these trusted components to provide isolated partitions for the use of data,
and to allow trustworthy operating systems to manage user input and output.

We have also spent considerable effort implementing the components of the
authority-mode SP architecture. Our new testing framework lets us run real trusted
code in a realistic environment with actual untrusted operating systems and software
applications running. We have instrumented this software environment in a virtual
machine to permit real attacks that have the same effects as truly compromised
software. We similarly instrument the virtual hardware to simulate hardware attacks
and observe hardware events. However, unlike in a real adversarial environment,
using the testing framework we have careful control over how and when attacks
occur; we are able to observe the results with fine granularity, all the way down to
the hardware level of processor registers and physical memory, and up to the level of
system calls and application interfaces.

We have used the testing framework to implement our emergency management
TSM. We test its interactions with the user and the remote authority, and we use it to
protect data, upon which we verify its implementation of access control policies. By
implementing the authority-mode architecture on the testing framework platform, we
have delved deeper into design issues for trusted software, such as how to use secure
memory in real code, leading us to a new design concept for Secure Areas.

Along the same lines, the integration work of the SecureCore project led us to
design virtualization support into the SP architecture. We needed to make the ar-
chitecture itself not be a source of vulnerabilities through covert channels, without
sacrificing the security of the hardware primitives that enable remote trust and revo-
cation. The result is a design that maintains security in both areas and adds support
for multiple virtual machines, each using the security features of the SP architecture
on its own data.

Overall, we have introduced a new hardware-software security architecture that
secures data against confidentiality and integrity threats in ways that are not possible
with a software-only solution. Our security architecture supports a variety of usage models and provides a root of trust for additional security systems based in software. We have demonstrated how to integrate our design with other security systems, such as the SecureCore separation kernel, and how to apply our architecture to additional platforms, such as embedded devices. Finally, we have implemented key components of the design to show its practicality and have created a new testing framework, which provides a platform for further development and analysis of our architecture as well as other security architectures.

8.1 Future Work

There are a number of areas for future work based on our authority-mode SP architecture, the testing framework, and the related implementations. Generally, these fall into the categories of: (1) additional design and implementation details that are needed to make our architecture commercially viable; (2) related problems that should be solved to provide a complete security solution but are not directly relevant to our designs; (3) related problems that address additional threats that we do not consider in our work; and (4) the application of the architectural concepts in our work to other platforms or processor models. In the rest of this section, we consider several of the important architectural components of our work and explain some specific ideas for each of these categories.

8.1.1 Trusted Software Modules

In our work, we develop example Trusted Software Modules (TSMs) for key management and for emergency management on the E-Device. The TSMs include a design for secure local storage of keys and policies, and they demonstrate how untrusted software can interface with trusted software. Additional TSM designs should be explored in future work. Our hardware places no specific limitations on what a TSM can do internally, how it can store and process data, what inputs and output mechanisms it can use, and what interfaces it can provide. Additional work can be done to design new application-specific TSMs that solve different problems than those we have addressed. New TSMs can also be written to interact with other software security mechanisms that could benefit from a hardware root of trust.

One method for designing new TSMs is to take existing software and divide it into security-critical and non-critical parts, designating the former as TSM code. This method should be applied to interesting applications with an emphasis on evaluating the security of the interface between the two parts and on how an adversary might attack the untrusted part to influence the operation of the trusted part or the interactions with the user of the system. Automating this process would be particularly interesting, and might derive from existing work on inverse taint analysis, where initial secrets are identified — such as keys, critical data, or user input — and code paths that access these secrets are considered critical. The hope would be to exclude large amounts of code that are not required for critical aspects of the user interface or
for the core components of the program that need to process actual secrets. Manual intervention and redesign of software may be required to assist this separation, in particular if the results of certain critical user actions that would be likely targets of attacks can be verified by trusted code, rather than including entire user interface modules in the TSM. For example, a banking application could confirm that the user intended to make a financial transaction, while leaving code for analysis and data presentation, which informs that decision, outside of the TSM.

In addition to designating certain code as trusted, the resulting TSM code needs to be modified to properly use secure memory for its secrets, and to use the on-chip master secrets as the root of trust for confidentiality and integrity. These would also be anticipated to be manual design decisions, which might be implemented with the assistance of automation tools. For new programs, programming language constructs and tags could assist compilers in the automatic generation of code using SP features. Rather than designate individual operations or memory regions as using secure memory, the programmer might want to tag individual variables or data structures as needing secure memory protection. The compiler would then ensure that they are secure whenever used in registers or memory.

Another component of TSM design requiring careful attention is the interface itself. While good interfaces should only permit safe operations, this may require the ability to authenticate calls made to the TSM. After a user is authenticated, it can be difficult to judge user intent when inputs are relayed through other software. Even if the user’s intent to activate a TSM feature is clear, the results of a request should only go to the intended application. Thus at a minimum, the TSM should be able to reliably identify which other software is making calls to the TSM interface. Operating system or some other support would be necessary to identify applications securely, protect the integrity of those applications, prevent impersonation, and then contain the data of the applications. Such mechanisms in the operating system would need to be protected and trusted. The process could also work in reverse where hardware or software hooks can be called by the TSM to dynamically evaluate its execution environment and that of other processes. The TSM would need to determine if the system is trustworthy in its current state, rather than having mechanisms to ensure that software is always secure. Hardware support for process isolation and virtualization can make this more reliable.

In our work, we make assumptions that the trusted software has been evaluated to be secure and trustworthy. Ensuring the security of the development process and the correct behavior of the resulting code is a separate problem from what we address in protecting the execution of the software. Our testing framework provides a platform for black box and white box testing of software, with the intention of testing both TSMs and the interactions between TSMs and other software. Additional work is needed to develop the criteria, tests, and attacks for this platform that can reliably determine if the software being tested meets its stated security goals with some level of assurance. This should be done in the context where source code is available and where it is not available. It is especially important to test how a TSM makes use of the SP hardware mechanisms to protect itself, and to find flaws in the underlying assumptions used. Analysis should be done on what assumptions are made
by programmers and compilers, in particular when using secure memory and when defining blocks of TSM code that might be called independently or out of sequence.\(^1\) There might be unexpected dependencies on internal TSM state or on the order of some operations.

Regarding development, TSMs should be treated like any other security program where certified development and testing processes are used to write code. New procedures should be developed to ensure proper design techniques are followed for TSMs. For example, assumptions are made that TSMs clear processor registers before exiting CEM and that secure memory is used for all intermediate data. These should be tested, likely using our testing framework, before a new TSM is deployed. Formal methods should also be considered in this context to produce code that can guarantee certain invariants or provide proofs of expected behavior, when given access to the invariants of the SP architecture.

As Trusted Software Modules are made larger, performance tradeoffs should also be re-evaluated. The use of on-chip caches helps to keep the penalty of our cryptographic operations low, and performance is only affected for the protected software and not other software on the device. However, frequent interrupts and switching in and out of trusted code will create some overhead in the software. For example, when a TSM protects its execution stack in secure memory, additional steps are required for each Begin_CEM or End_CEM operation. These operations are more frequent when the TSM must exit before each system call. TSM design practices might reduce this overhead, as would new SP-aware compilers or operating systems that provide TSM-based system call interfaces.

### 8.1.2 SP Architecture

In addition to defining new trust models and software for authority-mode SP, we have made specific improvements to the base SP architecture, including new secure memory and virtualization techniques. There are a number of areas for continued research on the design of SP and similar security architectures.

One area of future study is to apply our techniques for remote trust to other security architectures. Remote trust depends primarily on the binding of the master secrets in hardware to the trusted software. We take advantage of SP’s Concealed Execution Mode (CEM) to protect the trusted software and bind it to the device and the secrets; other methods of protecting trusted software could work as well and should be explored. For example, a software implementation of the remote trust mechanisms might be feasible on some platforms that use other means for binding to a device or verifying the system software. While the SP architecture focuses on a minimal design with symmetric cryptography and no burnt-in keys, other hardware

---

\(^1\)For example, a TSM will be composed of many blocks of code, each starting with Begin_CEM and ending with End_CEM. This permits execution to leave CEM to make calls to untrusted software or to interact with the input and output mechanisms. If these blocks are not used in the proper sequence, the effects of the trusted code may not be what was intended, due to assumptions made about the state of the system or of the data in secure memory. Therefore each block of TSM code must be written with additional checks to verify the current state and prevent these types of attacks.
designs could alternatively simplify the initialization process and allow more flexibility in upgrading trusted software, while still providing remote trust and reliable revocation. More generally, variations in the threat model may also lead to differing hardware and software requirements.

Within the SP architecture, we would like to expand the ability to identify and trust software other than the Trusted Software Modules. Rather than having a single level of trust for protected software, future work should explore separating a TSM into components at multiple trust levels. For example, the most trusted code would run as it does now in CEM with full access to the device secrets and secure memory, but some code might need protection from the operating system without access to all protected data. Thus an exploited bug in this less trusted code would have a lower risk of leaking critical keys and data.2 We’ve also considered providing only code integrity protection, without interrupt protection, secure memory, or device secrets, to larger parts of an application as a solution to identifying code that can make calls to TSM functions. Such a level with less protection mechanisms applied can improve performance compared to putting all of the components into full CEM protection. This code would also run without the constraints of CEM, such as the inability to make regular system calls.

Along the same lines, integrity checking alone would be appropriate for identifying a hypervisor kernel before it is given access to extra-privileged processor modes and instructions for virtualization on an SP device. When special hardware, such as secure I/O devices are present, these too could be restricted to only integrity-checked kernel code. Special signatures would distinguish each distinct level of trust in signed code for the Code Integrity Checking hardware.

Additional trust domains within an application’s TSM code could instead permit more modularization, where components are equally trusted but even trusted data should not flow between them, in order to mitigate damage if exploitable bugs are found in one module. Any of these changes to the levels of hardware protection require new security evaluation for the interactions between code at different levels and with untrusted software. They also require thought into new usage scenarios, design techniques for trusted software, threat models, and attacks.

If we consider the usage scenario of a device more broadly, designs are limited because SP currently permits at most one trust domain for all TSMs on the device.3 Similarly, authority-mode SP only supports a single remote authority. In practice it would be desirable to have the same trust available to multiple applications with mutually distrusting TSMs. The architectural methods we employ should be extended

---

2The second level of TSM code might run in CEM without access to derived keys or the SRH register. This might be used for libraries that are implement secure communications protocols or that analyze policy and implement access control decisions. Session keys, policies, and encrypted data could be passed in to these libraries as needed, rather than providing direct access to all critical secrets.

3While currently all TSMs must be mutually trusting, the use of policies attached to keys and data in the secure local storage allow for different TSM components to serve distinct purposes with effective access to only a subset of the secrets. For this to be reliable in the current model, all TSMs must enforce the policies uniformly.
to add this support. Some work in this direction has already been done in Bastion [29]. For authority-mode, we’d like to permit a device owner to use remote secrets from mutually distrusting authorities, or to have a single authority act as the device owner, but then install trusted software and secrets for multiple distrusting organizations.

Existing SP trust mechanisms should also be extended to a variety of processor designs. We have not yet explored the consequences of applying SP to multi-core processors. On that platform, some of the components should be replicated per core (e.g., interrupt registers, CEM state), while others should have a single copy per chip (e.g., Device Master Key, Storage Root Hash). Details such as access to cache tags and secure memory will depend on the specific implementation of the multi-core architecture. One must investigate concerns about concurrent access, in particular analyzing the security affects of having a trusted thread running on one core while an untrusted thread runs on a different core. Other complications are expected to occur, such as suspended threads being restarted on a different core, scheduling constraints, and operating system awareness of the use of TSMs. These become more complicated when there are also multiple trust domains, with distrusting TSMs running simultaneously on different cores, sharing some levels of cache.

Future research can also study supporting other types of remote trust. In this work, we consider accessing an organization’s data on a user’s local device. Instead, in grid computing scenarios, users want to securely use their own data on a remote computer; they do not want to completely trust the operator of that remote computer or server to protect their data or execute their code properly. Some of the basic mechanisms needed are very similar to SP, where a remote CPU executes trusted software with verification of confidentiality and integrity to a remote party, however different trust mechanisms are needed since users cannot physically verify the hardware and then install their own roots of trust. Initialization must be remote, and the remote server must prove that it is using a secure system that will correctly execute the users’ code without revealing any data or modifying the results. Public-key cryptography techniques, such as those used by TPM, may be necessary. This is also a case where the one-to-one trust relationship is insufficient, since the server will need to run trusted processes for many mutually distrusting clients.

This problem becomes even more complicated in cloud computing, where users’ data is put ‘into the cloud’, on a remote server, but will be used by software applications designed by the cloud service provider. Ideally users would like their data to remain contained and protected while being processed on the remote server and made available to them on demand over a network connection. Their data may need to be shared among multiple cloud services, operated by different organizations, over the course of processing even a single request for a service. The users would like to remain in control of the dissemination of their data even in this case. For example, if users have an online backup service for their personal computers, a significant quantity of their data may reside on the remote server. The provider can offer to index the data for quick search and retrieval, and may want to process some of the data in an automated system to show advertisements when users log in. Such a scenario is more palatable with regard to security and privacy if the users have some assurance that the provider cannot directly access their data and use it for other purposes.
8.1.3 System Software and Secure I/O Mechanisms

In order for a Trusted Software Module to be useful on a remote device, it must interact with the local user. The platform therefore needs to provide secure input and output (I/O) mechanisms such that the TSM can be confident that user inputs are legitimate, having not been forged or eavesdropped by an adversary, and that displayed data has not been modified before being presented to the user or copied by an adversary for misuse.

In our work we rely on trusted system software to provide secure I/O, however future work should explore how additional hardware mechanisms can assist in providing secure input and output. Critical components of any mechanism will address how to authenticate the source of data to the users or the destination of their inputs, how to enforce that only one software entity is the source or recipient of the data, and how to multiplex use of the display and input devices between this trusted entity and the rest of the software. The existence of such mechanisms would reduce the need to ever export sensitive data to untrusted software in plaintext. Once they exist, a further complication is to provide a rich user experience with a reasonable interface, while minimizing the amount of software required to render data and manipulate the display in order to avoid vulnerabilities.

We can also gain in overall security by better protecting and isolating parts of the system software itself. A TSM relies on the operating system for a number of services through system call interfaces. However, since the TSM cannot make system calls directly, there is an opportunity for untrusted software to manipulate the results or the parameters of such calls. To address this, future work should consider design practices for TSM code that are aware that such results are untrusted. Furthermore, while the TSM can generally detect attacks on the integrity of its storage and communications that might result from such manipulated system calls, it has no mechanism to recover from such attacks. Therefore, the reliability of the system can be improved significantly if parts of the operating system’s API could be accessed securely and have the results trusted.

Overall, the SP architecture is subject to many different denial of service attacks, which were not addressed in our threat model. For future work, protecting availability of data and services should also be considered, and is in addition to the protection of confidentiality and integrity. With parts of the OS protected by hardware or otherwise made trustworthy, then important components such as process scheduling, interrupt handling, virtual memory management, and filesystem access could be made reliable and reduce the opportunities for these denial of service attacks. A full study should be performed on the operating system components that are relied upon by trusted software and how they might be reconfigured to be within a trusted domain.

Similarly, we would like to further explore the work that we started for the SecureCore architecture on using hardware security mechanisms for mandatory access control and for a multi-level security device. There are many usage scenarios where having strict partitioning of trust domains is beneficial. Many of these can be seen in applications for virtual machines, with military and other MLS systems representing the most security critical of the uses. Data must be made available to users of the
device, with some ability for the user to share the data, while the consequences of accidental disclosures and data dissemination are severe.

Additional study is also needed in regard to how a user interacts with the device to ascertain its state and authenticity, especially given that each partition may have different levels of security. Future work should further study how the device protects the full software stack, how to create hardware and trusted software that are aware of when access is made to data that they do not directly control, and how users can be aware of the ways software accesses and uses their data without explicit permission.

8.1.4 Security Testing and Evaluation

For our testing framework in Chapter 5, future work can add capabilities to the framework itself, making it more flexible for testing new architectures. Additional work can be done to implement more of the core SP architectural features in the hardware emulation, including Code Integrity Checking and better support for interrupt handling.

Future work is needed to produce a better development platform for trusted software, to serve as a demonstration of the capabilities of the framework and to show that the type of trusted software used in the SP architecture is feasible to build. Future work should develop a model for writing TSM code, along with the necessary libraries, language tools, and development environment, such that programmers can be confident that certain security properties will be met and that the code will not be susceptible to known exploits and vulnerabilities. In particular, there are many methods of attack that are unique to the SP architecture, based on whether or not SP’s features are used correctly. The development infrastructure should make it easy to avoid these vulnerabilities. Once this infrastructure is in place, future work should develop tests in the framework that verify that the resulting code is not vulnerable to the relevant attacks and that the expected security properties still hold under the anticipated operating conditions.

Future work can also develop a few general-purpose structures and layouts for TSM designs, which can be reused and then easily tested for certain security properties. Our secure local storage structure is one such example, where we specify how data is stored by the TSM on disk and how that data is protected. In turn, a new TSM can simply use our data structures for its own data and policies and use a set of pre-written attack tools to verify properties such as confidentiality, detection of modifications and replay, and reliable revocation. Another example of such work in progress is the infrastructure for supporting a secure stack. The TSM is expected to initialize secure memory regions for use as secure stacks and change various stack pointers to take advantage of this region. Integration of these techniques with a compiler would be even more powerful and would provide more certainty that all critical data will be protected.

For the framework itself, we have implemented an emulation module for the authority-mode SP architecture, but it would be beneficial to offer more abstract hooks into the VMM, such that a specification for a new architecture can be provided and more easily implemented in the VMM, without modifying the VMM code di-
rectly each time. This would be accomplished with user-specified data structures for various types of simulated hardware registers, methods for adding instructions to the instruction set, and hooks into low-level hardware behaviors in instruction execution, cache behavior, memory access, I/O devices, etc.

Similarly, more hooks into software components would simplify the creation of attack scripts. Existing techniques for understanding operating system internal state should be integrated into the framework to provide attack script events at higher levels of abstractions. For example, we currently allow the framework to intercept system calls individually, but there should be a mechanism to watch for changes to a particular file on the file system. This would require filtering for the relevant system calls, analyzing parameters, and either probing the operating system’s data structures or maintaining parallel state to sufficiently understand each operation.

Another use of the work we have already done on the testing framework is to expand our SP architecture emulation module into a full software implementation of the SP architecture. Once the VMM implementation of SP is complete with all necessary security features and performance is optimized, the resulting virtualization platform will provide real protection of trusted software inside the virtual machines from guest operating systems. This serves as an even better testing and development platform in anticipation of building real processors with SP. It can also serve as a complete solution on platforms with certain threat models. If the hypervisor/VMM can be protected and trusted, then users can rely on their secrets being protected inside the virtual machines. This is most practical when physical attacks are not a concern or when another existing hardware mechanism, such as a TPM chip, can adequately protect the hypervisor (i.e., if offline physical attacks are possible, but physical probing cannot occur during operation).

8.2 Implications

Remote trust scenarios like the ones we discuss in this work will become increasingly prevalent as consumers store more of their data “in the cloud” on remote data centers and internet access becomes pervasive. Consumers will frequently access their personal and enterprise data from portable computing devices, such as smart phones and netbooks. These portable devices will therefore become increasingly attractive targets for attackers. We hope that architectures like those presented here can help consumers and businesses maintain the security of their data, without hindering the development of further innovations.
Appendix A

TSM Best Practices

Table A.1 lists some best practices that we have developed for writing TSMs securely using SP mechanisms. These should be used as a starting point for any TSM development. Thorough testing and security analysis are still necessary.

Table A.1: Best Practices for Developing Trusted Software Modules

<table>
<thead>
<tr>
<th>Type</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>After each Begin_CEM instruction</td>
<td>Before using any previously loaded secure data, verify that all previously defined Secure Areas are still properly registered in hardware and are at the correct location in memory with the expected size.</td>
</tr>
<tr>
<td></td>
<td>Create a new Secure Area for the TSM’s execution stack and swap the stack pointer to use this Secure Area. Copy any parameters into the secure stack.</td>
</tr>
<tr>
<td></td>
<td>Validate all input parameters and data that are accepted from untrusted code or as results of syscalls from the untrusted OS; place validated copies in secure memory or general registers.</td>
</tr>
<tr>
<td></td>
<td>Ensure that any necessary intermediate data or data from the secure persistent storage is loaded in the secure memory and has been validated against the SRH.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table A.1: (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before each <em>End CEM</em> instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write back any intermediate data or modifications to the secure persistent storage that is in secure memory or registers that should be permanent, updating the on-chip SRH as necessary.</td>
</tr>
<tr>
<td></td>
<td>Copy any output data from secure memory into unprotected memory or general registers. Copy any return values from the secure stack to the unprotected stack. Clear any secure data from the secure stack and remove the Secure Area that was used for the secure stack.</td>
</tr>
<tr>
<td></td>
<td>Clear and remove any other Secure Areas that are no longer needed by other TSM components.</td>
</tr>
<tr>
<td></td>
<td>Clear all general registers that were used for intermediate data, except registers that are intentionally being used to return data to caller.</td>
</tr>
<tr>
<td>Secure Memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All control flow decisions in TSM code should be made based on validated data in general registers or secure memory to prevent attacks by untrusted code.</td>
</tr>
<tr>
<td>Derived Keys</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When creating derived keys, always distinguish keys used for different purposes. Concatenate the nonces used to make multiple keys with constants that are unique prefixes for each purpose, such that the same key is never used for two different purposes (e.g., for both encrypting and hashing, or as both a storage key and a communications session key), even if the chosen nonce is the same.</td>
</tr>
<tr>
<td></td>
<td>Generally, one should never release derived keys to untrusted code or outside of the CEM-protected registers and secure memory. Such a breach could allow untrusted code to violate confidentiality or integrity properties of TSM storage or communications. If the <em>DRK Derive</em> function is ever used to create values that are available outside of the TSM, such as authentication tokens, and where untrusted inputs are used as part of the input function to the derivation function, be sure to prefix the user input with a constant value that is only used for keys that are revealed, and use a different constant for all other keys that are generated.</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Type</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Storage</td>
<td>When loading data from secure persistent storage, always check the integrity before using the data by verifying chain of MACs, including the root hash stored in the on-chip SRH.</td>
</tr>
<tr>
<td></td>
<td>Always check for attached policies when making use of data in the secure storage.</td>
</tr>
<tr>
<td></td>
<td>When writing back modifications to the secure storage from secure memory, ensure that the on-chip SRH corresponds to a valid copy of the secure storage on disk at all times. First write the updated secure storage to disk, keeping the old copy in tact. Second, update the SRH using the SP instructions. Finally, once the SRH update is successful, the old copy on disk can be deleted.</td>
</tr>
<tr>
<td></td>
<td>Always write back modifications and update the on-chip SRH before notifying the authority or a remote party that a policy-change, usage counter, or update to storage has been completed. If the SRH has not been updated, an adversary could still roll-back the change by replaying data on disk and a system reboot or application crash could cause changes to be lost.</td>
</tr>
<tr>
<td>Recovery</td>
<td>If an application tries to call a TSM function but the TSM is in use, the operation should be retried after some delay. If another process has a thread that is stuck in CEM and will likely never exit, that process must be killed and the CEM state cleared (including the CEM Mode and all general registers). This can be accomplished using <code>Save_SPRegs</code> and <code>Restore_SPRegs</code> as defined in Section 7.2.1 or using a new privileged instruction explicitly added for this purpose.</td>
</tr>
<tr>
<td></td>
<td>An application that contains a TSM might keep a backup copy of the signed TSM code and encrypted secure storage contents, such that these could be restored in case of an integrity failure. Restoring the secure storage requires that a copy is kept of each update, since only the last version that corresponds to the on-chip SRH will be valid.</td>
</tr>
</tbody>
</table>
Appendix B

Testing Framework
Implementation Details

Table 5.4 summarizes the API to the Event & Attack Module for hardware components. Table B.1, below, provides the detailed specification for the hypercall interface in the VMM that implements this API.
Table B.1: TF hypercall interface for kernel-mode and user-mode components. An API is provided for these calls in the TF Interface Library.

<table>
<thead>
<tr>
<th>Function</th>
<th>VM</th>
<th>Faults</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register_Events (long mask, int syscall_num)</td>
<td>SUT</td>
<td>none</td>
<td>Register with the Events &amp; Attack Module to have SP events forwarded to the TS Proxy and/or the TS Controller. If an IRQ is registered with Forward_Events() on the SUT, the event will be sent as a virtual hardware interrupt. Otherwise, if a syscall_num is specified during registration, SP hardware events will be forwarded as redirected system calls (simulated “int 0x80” instruction) with parameters in registers: EAX = syscall_num; EBX = 6 (a constant hard coded as AUTHSP_EVENTS_SYSCALL_CMD); ECX = Instruction hypercall number; EDX = SPFault code; ESI = Instruction parameter 1; EDI = Instruction parameter 2. The SP events are filtered by the specified mask (specifying SP Faults and/or SP Instructions). Set mask = 0 to disable forwarding. In addition, the filtered events will be forwarded to the TS, which can register with Forward_Events() to also receive the events as virtual hardware interrupts.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table B.1: (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>VM Faults</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward_Events (int</td>
<td>none</td>
<td>Register with the Events &amp; Attacks Module to have previously registered events and TF data messages forwarded to the guest OS kernel via the specified IRQ_num (or 0 to disable). On the SUT, forwards SP events that were previously registered with Register_Events(); on the TS, forwards SP events sent from the SUT and TF data messages sent via the hardware channel to the TS. When an event is to be forwarded, a virtual interrupt is generated by the VMM on one of the hardware IRQ lines, triggering the guest OS interrupt handler. No data accompanies the interrupt — the guest must subsequently call Get_Events() and Get_Data() to determine what type of event occurred and to receive the data. These calls should be made as quickly as possible, since additional events or data messages could occur which might overwrite the current event/data pending in the VMM, possibly overwriting that previous data. Both a data message and an event could occur during the same processor cycle on the TS with only a single interrupt being triggered.</td>
</tr>
<tr>
<td>TS/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
Table B.1: (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>VM</th>
<th>Faults</th>
<th>Description</th>
</tr>
</thead>
</table>
| SP_EVENT ← Get_Event (int clearEvent) |    |        | Retrieves the last SP hardware event that triggered an IRQ. New incoming events will overwrite the last saved, whether or not it was retrieved. The fields are returned via processor registers:  
|                               |    | none   | EAX = event type;                                                         |
|                               |    |        | EBX = SPFault code;                                                       |
|                               |    |        | ECX = Instruction hypercall number;                                       |
|                               |    |        | EDX = Instruction parameter 1;                                            |
|                               |    |        | ESI = Instruction parameter 2;                                            |
|                               |    |        | EDI = Event counter.                                                      |
|                               |    |        | The Event Counter increases sequentially for each event that arrives. An event can be optionally cleared from the VMM by setting the clearEvent parameter. This is optional in case multiple software components are involved in processing hardware events. If cleared, future calls will return 0 for all fields (except the event counter, which maintains its last value) until a new event occurs. If not cleared, the same event may be retrieved multiple times. |
| Request_Read (TF_DataTypes type, void *addr, size_t size) |    | Busy   | Read system state from the SUT. Request_Read() can access general registers, SP registers, physical memory, or virtual memory in the SUT, as specified by type. Reading memory requires that an addr and size be specified (size must be no more than one page — currently 4096 bytes). The result of the read will be returned asynchronously, raising an IRQ to be retrieved with Get_Data(). The returned data message will indicate success (with the data attached) or failure (e.g., if the VMM cannot read the requested data due to an invalid address or unmapped virtual page). Certain other faults are raised immediately if they occur: when inter-VM channel is in use (Busy), when the call is made by the SUT instead of the TS (Access), when the type is invalid (Invalid), or when the specified size is greater than one page (Invalid). |

(Continued on next page)
### Table B.1: (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>VM Faults</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF_Page_Fault ← Get_Data (TF_Data *data, int clearEvent)</td>
<td>TS none</td>
<td>Used to retrieve data that is being sent from the SUT in response to a Request_Read() call. Retrieves the last data message that triggered an IRQ by writing a data structure into virtual memory starting at the address specified by data. New incoming data messages will overwrite the last saved, whether or not it was retrieved. The data message can be optionally cleared from the VMM by setting the clearEvent parameter. This is optional in case a fault occurs or in case multiple software components need to retrieve the same data message. If cleared, future calls will return 0 until a new message arrives. If not cleared, the same data may be retrieved multiple times. If any virtual memory page needed to write the data structure is not mapped, the VMM cannot access the memory and will return a captured page fault in TF_Page_Fault, including the address of the faulted page. The caller should touch this page and retry the call. This retry process is handled automatically by the TF library. Note: If this call is made by the SUT, it will receive an empty data message; data messages are never sent to the SUT.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table B.1: (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>VM Faults</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request_Read_Bulk (TF_DataTypes type, void *addr, size_t size)</td>
<td></td>
<td>To speed up bulk reading of SUT memory, avoiding extra context switches on the host for each page of data, Request_Read_Bulk() can be used instead of Request_Read(). (Bulk reading is only available for physical and virtual memory, and cannot be used to access registers.) Rather than returning a single page of data, this operation causes a larger region of SUT memory to be written to a buffer file on the host system. Currently size is limited to 2GB, and both the addr and size must be page-aligned, raising a fault (Invalid) otherwise. Other faults are similar to Request_Read(). When reading from the SUT, any invalid or unmapped memory ranges will be saved as zeros. Request_Read_Bulk() will asynchronously return success or failure through a data message. If the operation is successful, the host buffer may be read from the TS. Using Read_Bulk_Data(), the TS retrieves data from the host buffer synchronously, copying data (currently up to eight pages at a time) from the given offset within the host buffer into TS virtual memory at buf (possibly causing a TF_Page_Fault). The host buffer is automatically deleted when subsequent bulk requests are made (even of zero size) or when the VM is powered off or suspended. Read_Bulk_Data() should not be used until the asynchronous response from the call to Request_Read_Bulk() is received. A ReadFailure fault is raised if Read_Bulk_Data() cannot read from the host buffer or if an error occurs during reading (in which case partial data may have been written to buf).</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VM Faults</strong></td>
</tr>
<tr>
<td>TF_Page_Fault ← Request_Write (void *buf, TF_DataTypes type, void *addr, size_t size)</td>
</tr>
<tr>
<td>TS Busy</td>
</tr>
<tr>
<td>TS Access</td>
</tr>
<tr>
<td>TS Invalid</td>
</tr>
<tr>
<td>Send_Command (TF_Commands command, unsigned long param1, unsigned long param2)</td>
</tr>
<tr>
<td>TS Busy</td>
</tr>
<tr>
<td>TS Access</td>
</tr>
<tr>
<td>TS Invalid</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>TF Page Fault SPRegs_Get (AuthSPRegs *regs)</td>
</tr>
<tr>
<td>TF Page Fault SPRegs_Set (AuthSPRegs *regs)</td>
</tr>
</tbody>
</table>
# Appendix C

## Acronyms

Table C.1: List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>An interface offered by a software component through which other components can access data and call software functions.</td>
</tr>
<tr>
<td>CEM</td>
<td>Concealed Execution Mode</td>
<td>A new processor mode in the SP architecture that protects the confidentiality and integrity of the execution of a Trusted Software Module.</td>
</tr>
<tr>
<td>CIC</td>
<td>Code Integrity Checking</td>
<td>A component of the CEM protection of the SP architecture where TSM code is signed using the device key and its integrity is verified dynamically during execution.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>The main component of a computer that performs processing and executes code. Often used to specifically refer to the microprocessor chip itself.</td>
</tr>
<tr>
<td>DAC</td>
<td>Discretionary Access Control [44]</td>
<td>A type of access control where access to objects is determined by the user’s identity. Typically each object has an owner who can control the permissions on the object.</td>
</tr>
<tr>
<td>DMK</td>
<td>Device Master Key</td>
<td>A non-volatile on-chip register for user-mode SP. Used to implement CEM, similarly to how the DRK is used for authority-mode SP.</td>
</tr>
<tr>
<td>DN</td>
<td>Directory Node</td>
<td>One type of node in the secure local storage structure defined for key-management in authority-mode SP. See Section 3.4.4.</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRK</td>
<td>Device Root Key</td>
<td>One of the two non-volatile on-chip registers for authority-mode SP that store root secrets. Used to sign TSM code, to protect secure intermediate data in CEM, and to derive keys for use by the TSM.</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
<td>The use of software or hardware components to control access to digital content and enforce access control policies on its use.</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
<td>A form of non-volatile memory used in an integrated circuit that can be repeatedly erased and reprogrammed.</td>
</tr>
<tr>
<td>ID</td>
<td>Identification number</td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td>Input and output mechanisms of a computer.</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt Request</td>
<td>A hardware signal, usually raised by an I/O device or peripheral, indicating that a device needs the attention of the processor, triggering an interrupt to handle the request.</td>
</tr>
<tr>
<td>L1</td>
<td>Level-1 cache</td>
<td>The first level of on-chip cache memory.</td>
</tr>
<tr>
<td>L2</td>
<td>Level-2 cache</td>
<td>The second level of on-chip cache memory. Typically this is the last level of on-chip cache memory and interfaces with off-chip memory.</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
<td>A value used to authenticate a message, verifying integrity and authenticity. It is generated by a keyed cryptographic hash (e.g., [131]), and the key is required to generate or verify the MAC.</td>
</tr>
<tr>
<td>MAC</td>
<td>Mandatory Access Control [44]</td>
<td>A type of access control where access to objects is determined by certain security attributes of the object and the subject. These security attributes are set by an administrator and typically cannot be changed by the subject (the user).</td>
</tr>
<tr>
<td>MLS</td>
<td>Multi-Level Security [17]</td>
<td>A specific type of mandatory access control policy where data is assigned a security level and users are assigned clearance levels, which are compared according to pre-determined rules to determine if access is permitted.</td>
</tr>
</tbody>
</table>

(Continued on next page)
Table C.1: (continued)

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>An OS directly manages hardware resources (in a non-virtualized system) and provides services to applications through shared device abstractions and APIs.</td>
</tr>
<tr>
<td>RDN</td>
<td>Root Directory Node</td>
<td>A special type of directory node (DN) that sits at the root of the secure local storage structure in authority-mode SP(^–). See Section 3.4.4.</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
<td>An integrated circuit that combines the microprocessor, caches, main memory, and sometimes other components onto the same physical chip. SoCs are frequently used for embedded systems.</td>
</tr>
<tr>
<td>SP</td>
<td>Secret Protection</td>
<td>The Secret Protection architecture [102, 51] is our hardware-software architecture, which uses hardware roots of trust to protect the execution of Trusted Software Modules and the confidentiality and integrity of keys and sensitive data as they are used by that software.</td>
</tr>
<tr>
<td>SRH</td>
<td>Storage Root Hash</td>
<td>One of the two non-volatile on-chip registers for authority-mode SP that store root secrets. Provides a small amount of on-chip replay-resistant secure storage. The SRH is typically used by a TSM to store the root hash of a persistent storage structure.</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
<td>One of the components of the testing framework (Chapter 5 that simulates the device being tested. It runs a full commodity operating system with applications and is subject to real attacks by the Testing System.</td>
</tr>
<tr>
<td>TCB</td>
<td>Trusted Computing Base [44]</td>
<td>The set of all hardware and software components that are trusted in a system to support the security policies.</td>
</tr>
<tr>
<td>TF</td>
<td>Testing Framework</td>
<td>Our new Testing Framework, as described in Chapter 5.</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security [45]</td>
<td>A cryptographic protocol used for secure end-to-end network communications (e.g., [57]).</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TML</td>
<td>Trusted Management Layer</td>
<td>The lowest layers of the SecureCore software stack [90], referring to the combination of the least-privilege separation kernel and the trusted services layer. Together these serve as the hypervisor for hosting guest operating systems in the various partitions.</td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
<td>A specification [174] for a commodity hardware chip often added to the motherboard of personal computers.</td>
</tr>
<tr>
<td>TS</td>
<td>Testing System</td>
<td>One of the components of the testing framework (Chapter 5 that manages the testing process; the Testing System controls and observes the System Under Test.</td>
</tr>
<tr>
<td>TSM</td>
<td>Trusted Software Module</td>
<td>The software module(s) in the SP architecture that execute with CEM and CIC protection and have access to the hardware roots of trust.</td>
</tr>
<tr>
<td>UMK</td>
<td>User Master Key</td>
<td>An on-chip register used as one of the master secrets for user-mode SP [102]. The value is generated by secure I/O hardware from a user-entered passphrase, and is then made available to the TSM to decrypt and access the user’s keychain.</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine [140, 69]</td>
<td>An abstraction of the physical device hardware that is controlled and emulated by a hypervisor, and typically behaves the same as the real hardware from the perspective of the software within the virtual machine.</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor [140, 69]</td>
<td>A hypervisor that provides virtual machine abstractions, allowing multiple operating systems to run concurrently on a single computer.</td>
</tr>
<tr>
<td>VMX</td>
<td>–</td>
<td>A particular component of the VMware [182] virtualization architecture. A VMX process runs in the host system for each virtual CPU of the virtual machine, and interfaces with the VMM for that virtual CPU.</td>
</tr>
</tbody>
</table>


dynamic information flow tracking. *ACM SIGOPS Operating Systems Review*,

applications and operating systems configurable. In *Proceedings of the 7th sym-
posium on Operating systems design and implementation*, page 292. USENIX


In *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies
for the Protection of Intellectual Property in the Network Multimedia Environ-

[176] International Telecommunications Union. ITU-T Recommendation X.509, The

[177] N. Vachharajani, M.J. Bridges, J. Chang, R. Rangan, G. Ottoni, J.A. Blome,
G.A. Reis, M. Vachharajani, and D.I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th an-


[179] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability
procedures in the formal verification of superscalar and vliw microprocessors.
In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages

sjsu.edu/faculty/stamp/students/RamyaVenkataramu_CS298Report.pdf.

on wireless sensor network devices. In *IEEE Conference Emerging Technologies


security: A survey*, chapter 17, page 367. Security in Distributed, Grid, Mobile,


