UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Embedding Security into Systems After Their Design

Permalink
https://escholarship.org/uc/item/1vn6v7wg

Author
Capelis, D J

Publication Date
2015

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EMBEDDING SECURITY INTO SYSTEMS AFTER THEIR DESIGN

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

D J Capelis

September 2015

The dissertation of D J Capelis
is approved:

Darrell DE Long, Chair

Ethan L Miller

Ahmed Amer

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**
- 1.1 New Possibilities . . . 2
- 1.2 Ease of Deployment . . . 2
- 1.3 Consolidation . . . 3
- 1.4 Consistency . . . 5

2 Changing Networking **7**
- 2.1 Related Work . . . 10
- 2.2 Technical Detail . . . 13
  - 2.2.1 Core Services . . . 13
  - 2.2.2 Session Initiator . . . 21
- 2.3 Performance . . . 21
- 2.4 Potential Improvements . . . 24
  - 2.4.1 Integration into the Kernel . . . 24
  - 2.4.2 Integration into Hardware . . . 25
- 2.5 Deployment . . . 25
  - 2.5.1 Deploying with Unmodified Applications . . . 26
  - 2.5.2 Deploying with Unmodified Computers . . . 26

3 Changing the Computer Organization **28**
- 3.1 Goals . . . 29
- 3.2 Related Work . . . 31
- 3.3 Technical Design . . . 33
  - 3.3.1 Trusted Loading . . . 35
  - 3.3.2 Trusted Data . . . 37
  - 3.3.3 Trusted Runtime . . . 39
  - 3.3.4 Trusted Channels . . . 45
  - 3.3.5 Designs for Trusted Networking . . . 46
- 3.4 Designing Interaction with Users . . . 50

3.4.1 Experimental Design . . . 50
3.4.2 Data . . . 53
3.4.3 Findings . . . 54
3.5 When Encryption is Not Enough . . . 56
3.5.1 Proof of Concept . . . 58
3.5.2 Tools for Temporal Cryptanalysis . . . 58
3.5.3 Would You Like To Play A Game? . . . 59
3.5.4 Game Over: Attacker Wins . . . 60
3.6 Implementation Concerns . . . 61
3.6.1 Hardware Cost . . . 61
3.6.2 Code Analysis . . . 61
3.6.3 Deployment . . . 63

4 Final Thoughts **65**
4.1 What Allows Change? . . . 65
4.1.1 The Problem of Security . . . 66
4.1.2 Beyond The Threat Model . . . 67

Bibliography **69**

Abstract

Embedding Security into Systems After Their Design

by

D J Capelis

Security is rarely designed into systems and architectures from the beginning. Typically, security enters into the design process only after applications are built and security issues arise. While security is often dependent on specific use cases, decades of development provide an opportunity to synthesize common security needs into a set of critical features and embed them into the core underlying systems.

The advantages of doing so are fourfold:

1. Security features need only be implemented in one common place. Instead of each application implementing its own security features—often complex and difficult code where small bugs can result in large failures—only one implementation needs to be security-critical.
2. Instead of many application specific interfaces, applications present compatible and consistent security configuration implementations. Greatly reducing the difficulty of configuring, maintaining and understanding security policies across a system or network.
3. Instead of spending limited developer time on implementing the same security features in every application, security mechanisms integrated into the system reduce the burden on the application developer. Further, the quality of these mechanisms can exceed what a specific application developer would otherwise chose to invest their time in building for their own application.
4. By expanding the mechanisms available, some features which would otherwise be impossible to implement, become possible.

I explore this approach in two of the most fundamental areas of computer science: the network and the computer organization. Both of these areas provide decades-old interfaces with long-standing security needs. The applications which have grown on top of

these systems are mature and represent a range of needs, development effort, developer skill and solutions. This provides a diverse range of software to distill new security features from.

During my research, I embedded authentication, session encryption and role-based network visibility and access control into the Internet, reviving the concept of a session layer. Using the knowledge we have today, this set of primitives allows applications to solve problems that weren't as relevant when the networking community first examined the session layer. This foundation reduces the complexity of writing working, secure and authenticated network services.

I also explored architectures which provide the ability for an application to store data in a manner that is not vulnerable to interception and compromise by management software (the operating system, a hypervisor, etc) running on the machine. This capability does not exist in systems today and would allow more resilient systems to provide limited security assurances even in the case of compromised management software.

In this document, I show embedding security into existing systems yields tangible benefits over building solutions on top of systems without altering the underlying status quo.

Acknowledgments

With daily news of new governments and organizations using technology in more new ways to restrict freedom and civil liberties across the globe, computer security has been a dark field. It is filled with fun interesting problems if you can ignore the implications of those problems, but I've never been able to. The reality is many of the solutions to our problems are non-technical. We navigate a world of grey, knowing only how we hope our work will impact the world, never quite how it will. Without many wonderful and supportive people in my life, this dissertation would not exist.

Various communities of creatives, intellectuals, artists and hackers have inspired my growth over the years. The many amazing teachers through my life, those with titles and without, have given me gifts of untold value. The art group I participate in, Ardent Heavy Industries, provided the necessary distractions. To those who were there for me when no one else knew how, and to those who were there for me when I couldn't be: thanks. And of course, my mother, whose contribution to my existence is most literal of all. Due to her efforts, I was privileged with an amazing environment to grow from, make mistakes in and learn.

My presence in research is due to many supportive individuals: Ahmed Amer, who among many supportive acts once calmly helped me see the value of my work when I struggled to justify spending time on research while police were shooting protestors on the streets with teargas; and during the Arab Spring as an Egyptian, had the credibility needed to make that argument. Meredith, Sergey, Anna and the LangSec cabal who kept me connected to security. And without a doubt, my advisor Darrell Long, who has shown extreme patience and support for my work. He and others have been a key part of procuring the many exceptions to university policy that have comprised a necessary ingredient in my formal education. Darrell, along with Alex Orailoglu, who mentored and advised me extensively at UCSD when I was an undergraduate student; Ethan Miller, our lab's co-director; Tom Kroeger; and Jose Renau formed my advancement committee. Darrell, Ethan and Ahmed teamed up to form my dissertation committee, a group of amazingly supportive people who have asked only that I give them an excuse to give me a PhD. I hope this document provides them with just the thing.

To the people I forgot: I didn't. Thank you.

To time—and the change it brings.

Chapter 1

Introduction

Security is a hard problem, often difficult to describe in a rigorous fashion, much less easily quantify. Market forces usually dictate that what works now is more important than what works well. These structural issues mean the design of security technologies most often only occurs after the need becomes obvious. As security concerns are so often use-case specific, security is seen as an application concern, not a fundamental one. The result: many specific security problems are seen as solved in theory, but the lack of security as an integral part of system-level design have led to continued security issues in practice.

In this dissertation, I explore two areas of computer science with the aim of migrating security responsibilities away from applications and into the underlying systems they run on. For some of these security issues, existing techniques provide a solution, but the deployment of these technologies is far from ubiquitous or the interface to use them is far from simple. Other security issues can be solved by technologies that exist today, but are solved in a piecemeal approach in which every specific use case opts for different and frustratingly incompatible security technologies. Finally, some technologies provide solutions to security issues which cannot be solved on existing systems, no matter how the existing primitives are used.

1.1 New Possibilities

Perhaps most critical are those features which cannot be implemented on top of existing core technologies and require change to exist. Providing new ways to secure software that are not available in current systems is what advances forward not only security, but our definition of the types of tasks computer can do securely.

There are two features I focused on that the current architecture cannot support:

1. The ability for a network to make access control and visibility determinations based on *who* a user is as opposed to which IP address happens to be making a request. Current layering relies on applications to construct a user's identity. Lower layers cannot use that information and access determinations are based on the identity of the requesting machine instead of the identity of the person at the keyboard.
2. The ability of an application running in an operating system to protect data from compromise even when the operating system and other management software cannot be trusted. The current architecture provides the operating system with complete privilege over applications running within its environment. An application which runs on a computer is required to trust the security of the operating environment, an aspect that it and sometimes even its user may have little control over.

Adding these types of features is not possible without revision to the fundamental core technologies which compose our systems. Between enhancing security on a system-wide level and the possibilities these features provide in terms of changing how security is done and the types of activities that would become possible, revision of core technologies is necessary.

1.2 Ease of Deployment

Application support is one of the most common barriers to deploying security technologies. Application developers aren't always aligned with the security needs of

their users. Even when the developers agree that a security feature should be implemented, the development team does not always have the skill, time or resources required to implement a feature. Finally, when a feature does get implemented, developers often stop as soon as a minimal level of security has been achieved. For instance, it is rare for an application to add support for authentication mechanisms beyond a username and password, even though research [58] has shown that passwords are not a particularly good authentication mechanism. The two decades it took to bring modern security practices to NFS [63] provides a good example of how difficult it can be to deploy security beyond a minimal level.

However, once standard security primitives are available in a lower layer, applications support security features that are as advanced as the underlying framework can support. Most importantly, integrating security features into a lower layer often results in requiring *less work* for a developer to implement a feature than if that developer were to write a basic level of functionality themselves. This leads to wider deployment of security technologies and increasing levels of sophistication and flexibility.

Further, structural changes to an architecture allows a series of important security features to be implemented and easily available for applications. The incentive for code libraries run towards doing one thing very well. This means application developers using a library that accomplishes one specific task must use a different library to accomplish another security task. By incorporating a series of features into one fundamental layer, the marginal effort to incorporate each additional security feature drops as the base reference set for what the lower layer accomplishes expands.

1.3 Consolidation

From a software engineering standpoint, while security issues can be produced by bugs in a wide range of code paths, bugs in code which provide security features have a greater tendency to directly and immediately result in an exploitable security issue. Even worse, since "security" is hard to formalize and define, tests to determine whether security code is functioning properly are frequently difficult or even impossible to write. This means not only do the bugs in these sections of code tend to be high impact, but they're also difficult to find using standard code testing methods.

To mitigate this danger, many projects have dedicated teams of security experts that conduct audits and review changes for security-critical code sections. The outcome of this practice varies widely from team to team, depending in large part on the level of security expertise available and the priority of security issues within the project's management. For small projects, this type of review can be prohibitive and even large projects are forced to invest resources in focusing on security issues which may be better invested in providing a more compelling product. The outcome: security critical code sections slow down development progress.

This means projects are well served by writing as little security-critical code as possible. One option of doing this is simply not to implement that many security features, a surprisingly popular option which is often determined by externalities, development culture, the level of security in competing products and market forces rather than technical or security concerns. The other option is to share as much security critical code as possible so the expense and cost of review is amortized over as many projects as possible.

The most common mechanism to share security critical code between applications is to implement security code into programming language libraries. Unfortunately this leads to different applications using vastly different APIs, depending on the programming language of the code, the frameworks available on the system and the portability of the respective components. In addition, many applications use only a small set of features within a large and complex API. For instance with SSL/TLS [24], one of the many security issues this work impacts, OpenSSL [19] is one of the best known cryptographic libraries. It is frequently used to set up TLS and SSL connections. However much has been written about how frustrating it can be to produce working code with this library [49].

While this may seem like merely an implementation issue, alternative libraries are plentiful and frequently suggested [2] by practitioners. Some have met moderate success and made commendable improvements, the real issue is lost: the libraries are all trying to layer security on top of the system instead of integrating it in the layer it belongs. The complexity of these libraries are in part due to a structural lack of these security features in the core stack.

Not only will solving this structural defect likely lead to easier and better support for these types of security features, it will help migrate the responsibility away from applications to a lower level of the software stack. A system running multiple security-critical code sections all trying to do the same thing is as strong as the weakest code section. Allowing all the applications to use one highly reviewed and well considered code section or feature could reduce the system-wide risk profile considerably.

1.4 Consistency

From an individual application's perspective, a lack of consistency is not that large of an issue. The security protocols themselves are often standardized to allow interoperability. Unfortunately on a system-wide level, the various applications, frameworks and libraries implementing various security-critical functions using various levels of abstraction produce a remarkably frustrating experience. Configuring a system-wide security policy becomes extremely difficult, with either complex policy management tools which try to apply a unified policy across the system or a large amount of duplication as each system is configured separately.

For instance, one example of this complexity is restricting the use of network services to certain users. The mechanism to accomplish this is typically different for every service on the system. It is not uncommon for services to use their own independent user database for authentication, separate from any others on the system. Even those services which use the standard system user database have their own unique syntax, configuration files and mechanisms for specifying authorization. To some degree, authorization must be an application issue, but basic restrictions on who should be allowed to access a service is an issue every network service shares and every administrator must concern themselves with. And for this, there is no fundamental access layer beyond inflexible firewall control using IP addresses.

Integrating security features into lower layers allows for usable system-wide policies. This provides administrators the tools they need to give specific users abilities on the system without fear of exposing a large attack surface to the rest of the world. In addition, the consistency of having security functions in the underlying software layers provides users of the machine with a uniform experience for interacting with

security systems, lessening the cognitive burden of remembering the various steps and incantations different software services require of users before recognizing their identity and allowing them access to data and resources.

In the network, I use a prototype called *fived* to discuss the consistency and consolidation benefits of integrating identity into the network. I also discuss increased consistency for network mobility, encryption, service discovery and distributed identities, among others. In the computer organization, I outline hardware changes required to provide resistance against complete operating system or hypervisor compromise. All of these changes require fundamental change or addition to how these systems operate.

Chapter 2

Changing Networking

On today's network, an unwieldy array of different components are tasked with security responsibilities. Application developers routinely make mistakes in their security critical code, leading to bugs that manifest as worms and malware. Access control mechanisms on the network typically rely on *where* a user's computer is located, not on *who* that user is. The systems that authenticate users remain separated from the firewalls tasked with controlling access to various network services. The network is without the information required to make intelligent access control decisions. These problems are compounded by the Internet's remarkable resistance to change. Many security technologies have failed to achieve adoption over the years. *Fived* is a design for a unified session layer that integrates security features into the core of the Internet, one user, one network or one application at a time.

Let's begin with the problem of access control. Firewalls, the main source of access control in most deployed networks, dictate access control policies based on the host's IP address. Any network that wishes to support legitimate users' ability to access services from networks not directly owned by the organization must support a mechanism to bypass the security perimeter; this is typically provided in the form of a Virtual Private Network (VPN) [31] connection. Likewise, organizations that wish to offer courtesy access to guests have to institute registration processes and network partitioning just to allow visitors to check their e-mail without exposing internal services. In more complex organizations, where users have various levels of access, clearance or affiliation, network partitioning can get even more complex, arduous and brittle.

A session layer design provides the user with the notion of a session, but more importantly, allows them to authenticate and open up the range of services available to them. This layer makes access control decisions based on who the user is, what groups or roles that user may be a part of and any number of additional policies the network administrator might wish to support. This provides the type of comprehensive access control modern networks need.

Yet the benefits of a robust session layer extend beyond simplifying the lives of network administrators and reducing the complexity of security configuration. The current Internet architecture forces each individual network application to write large amounts of sensitive code to provide security features, including authentication and encryption. Applications often simply omit some of these features, while the remainder provide a wide array of encryption and authorization solutions of varying quality in terms of usability or security. A session layer puts security features on-par with core networking concepts like congestion control. With a session layer in place, applications can take advantage of one unified codebase to perform these types of sensitive operations. Support for new authentication mechanisms, new encryption technologies, or other new security features, can be added in one place and made immediately available to all applications running on the session layer.

One of the hardest problems of changing the Internet is adoption. Worse, in the next several years, new architectures which seek to gain acceptance on the Internet face several specific challenges. IANA has allocated their last free block of IP addresses [38] to the Regional Internet Registries and each has put in place emergency procedures to manage their last remaining IPv4 addresses. Over the next years, the Internet will reel as it reactively deploys the solutions networking researchers converged on 15 years ago. Due to this unfortunate timing, any realistic deployment of new networking technologies reliant on commercial network operators to adopt new equipment, standards or practices will be challenged for several years to come. Yet the switch to IPv6 will not solve many of the pressing problems that have become more apparent over the last decade and a half since IPv6 was designed. The Internet can't wait another 15 years for new technologies to reach deployment.

In an environment where developing a compelling improvement is merely a

necessary, but not sufficient condition for deployment it is critical that any proposed changes provide a realistic transition plan. *Fived*'s key deployability advantage is that it follows the end-to-end principle [56] allowing progress to trickle in from the edges of the network to the core. Potentially lethargic core network operators do little to harm the adoption of *fived*. Another key component of a deployment plan is that the system must not require a large critical mass before organizations begin seeing benefits. Users of *fived* gain some benefit right away. Downloading code is enough to allow users to start controlling access to their internal services. A set of compatibility libraries, layers and runtime tools can provide users the ability to obtain advantages from *fived* even before applications are adapted to interact with the session layer natively.

The final defining feature of a viable architecture change is its ability to lead us away from the current level of stagnation on the Internet. A new architecture change not only needs to overcome its own deployment challenges, but should attempt to open up the Internet in ways that allows increased flexibility in the future. The Internet's resistance to change isn't sustainable and without modifications, the network will not be flexible enough to head toward the future. Future technologies must be designed so if they succeed, the Internet can absorb new ideas, innovations and technologies at a higher rate than the current network.

The session layer as implemented in *fived* is extendable. Creating a new service can be as simple as picking a name for it, adding some lines to a configuration file and writing as little as 10 lines of code. This architecture is considerably more flexible than the fixed size headers most of our modern Internet protocols use today. The core services in this document should be useful for years to come, but future researchers can experiment with new variants on these services merely by adding it to their local session layer and beginning to use it. Standards can spread either organically or via vendors working together in a formal process. Innovative ideas can be demonstrated easily and adopted quickly as consensus develops.

In addition to ensuring that our own additions are flexible, *fived* aims to ensure that the underlying components of the current Internet grow easier to replace. This is the job of the session initiator, the component that establishes a session. The session initiator resolves names to addresses, deals with various transport protocols and sets up

connections or an ability to send datagrams in future networks. Adding a new transport protocol or changing Internet addressing merely requires modifications to the session initiator and any application using it can adapt. This allows the layers underneath the session layer to change and accept new technologies as well.

Fived provides solutions to a broad range of recent problems, with specific focus on embedding trustworthiness into the fabric of the Internet. The design has a transition plan, a design which allows for a suite of compatibility tools and has the potential to bypass many of the roadblocks during what's likely to be a messy transition process to IPv6. Finally, the design serves as a catalyst for past, present and future technologies by ensuring that if *fived* is successful, the deployment barrier on the Internet is reduced.

2.1 Related Work

One of the challenging parts about explaining *fived* has been in comparing it to existing technologies. *Fived*'s design goals revolve around incorporating solutions to problems where we know the network has needs. The solutions *fived* implements often aren't particularly different than existing technologies. It isn't in the choice of encryption algorithm or the protocol that *fived*'s contributions are really understood. It is in the way this session layer enables the use of these technologies in a way that creates a coherent architecture between every application using *fived*. It is in the way that *fived* shifts the responsibility into the underlying layers and eliminates sections of security critical code required in many of today's applications. It is in the way *fived* strictly adheres to the end-to-end principles and eschews any requirement that the core networking hardware know about our protocol for it to succeed. It is in the way the session layer architecture enables application access to these technology in a uniform way across the deployment base.

Which isn't to say no other projects have had these goals. Service-oriented network designs, such as those seen in Planetlab [17] and GENI [50] often have similar design goals. Chandrashekar's paper on a Service Oriented Internet [13] comes up with a strikingly similar design in some respects. In this paper, a session layer with a service-oriented architecture is fairly clearly proposed and outlined. The main difference

between these works and those of *fived* is a difference in how these systems interact with legacy technology. Many of these designs fall under the category of “clean slate” networking architectures, where the goal of the research is to clean up the Internet and switch to a “better” architecture. *Fived* on the other hand, is what I like to call a “dirty slate” design. The goal of *fived* is to add the features into the *existing* network that seem to be missing. When there's a way to do it that seems to prod the network towards a cleaner architecture, *fived* takes the opportunity, but the guiding goal is to get the features into the network. The resulting systems turn out fairly different.

On a feature by feature basis, there are many comparisons between *fived* and other systems:

Service discovery allows a computer to query whether a service is running. Traditionally this is done via attempting to establish a connection on a standardized port number and assuming that if a service exists on the machine, it will be listening there. *Fived* allows a user to instead specify services using a name, a minor improvement to usability that shifts the namespace from numbers to characters. Other software that has tried this approach includes the portmap [66] service, which protocols like NFS [67] rely on.

Encryption on the Internet is hardly a new feature, SSL [29] and later TLS [24] have been providing encryption services on the network for awhile. Newer protocol proposals, like MinimaLT [51], CurveCP [12] and QUIC [70] offer transport security with improved cryptographic properties. *Fived* implementations can include any or all of these protocols. Since the session layer is application protocol agnostic, it's transparent to layer 7 applications. This is similar to how Stunnel [76] or SSH tunnels [78] work. Those tools, of course, have few ambitions beyond providing transparent encryption.

Authentication and access control on a network level is currently a problem solved by a combination of VPNs [31] and firewalls [14]. Surprisingly, existing networks have made these technologies work from time to time, but it seems not unreasonable to point out that in practice networks experience problems using access control technologies which only make decisions based on what number a user's

computer currently is assigned by their network. A VPN exists on most networks to allow an end-user to borrow a number from another network when the one their computer has doesn't allow them to access the resources they want. *Fived* on the other hand, ties traffic streams directly to a user's identity and uses that to make access control decisions.

Network mobility is a feature which allows a device to switch underlying network transports without breaking their network connections. This feature is used today in cell phone networks [41] where devices roam between cells routinely and so mobility is built into the low-level network protocol. OpenFlow [47] is another system which has mobility features, allowing devices to move network flows from endpoint to endpoint. Both require extensive levels of support in the networking hardware. *Fived* implements these features in the session layer.

Stream multiplexing has become common again with the introduction of HTTP/2.0 [10] which formalized SPDY's [9] approach of multiplexing multiple HTTP streams over a single TCP connection. *Fived*'s multiplexing is somewhat different, since it is protocol agnostic. This allows any traffic between two endpoints to share a transport stream.

Virtual hosts allows application protocols servers to host more than one hostname on the same IP address. [42] This technique is present in some application protocols, like HTTP and SMTP, but is not uniformly deployed through the network. Generally, when an application connects to a port on a host, the service is not given hostname information to allow it to appropriately determine which content to send. *Fived* introduces protocol-agnostic virtual hosting by enabling the hostname the user specifies to alter service routing in the session server.

Distributed identity has increased in prevalence since the launch of OpenID [53] a decade ago. Since then, many large web companies have shipped their own incompatible distributed identity systems, from Facebook [28] to Twitter [74] to Google [33] there's as many different identity protocols as there are companies that want to control identity information. *Fived* also includes a light-weight distributed identity protocol, which allows users to use an authenticated session to prove their

identity to third-parties. One difference with *fived*'s protocol however, is that it eliminates the direct communication between the third party identity consumer and the identity provider, thus allowing for distributed identities that don't require users reveal to the identity provider where they're using the identity.

2.2 Technical Detail

2.2.1 Core Services

The following core services are the primitives I've selected to put into *fived*'s default set. *Fived*'s session layer protocol is loosely derived from the *tcpmux* protocol specified in RFC 1078 [45], from 1988. The basic *tcpmux* protocol is simple and can be implemented in under 100 lines of C. Each of the core services *fived* adds take anywhere from tens of lines of code to several hundred lines of code. These services work together to provide a broad range of session services. The essential features include service multiplexing, role-based authenticated access control, transparent session-wide encryption, mobility, virtual hosting and distributed identities.

Let's examine each of the core services in depth:

2.2.1.1 LIST

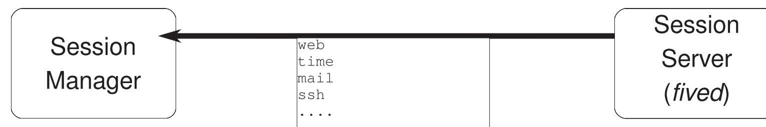


Figure 2.1: LIST Command

LIST enabled service discovery. LIST outputs a multi-line message which must be a list of the service names of the supported services, one name per line.[45] "Supported services" means services which the user is able to access. Services with restrictions only appear in LIST after a user has authenticated themselves with an authorized set of credentials. When LIST is followed by a service name, *fived* returns the service name if it exists. These two modes allow for dynamic service and extension discovery.

≫**LIST**≪

LIST

MULTIPLEX

TLS

HOST

http

Example Usage: A typical use of the LIST command on a *fived* daemon which supports a the core features LIST, MULTIPLEX, TLS and HOST as well as a service called "http."

2.2.1.2 AUTH

AUTH allows a user to authenticate a session. The exact mechanism to do this is server-specific as each organization tends to have their own requirements for credentialing users. The current *fived* prototype uses the Pluggable Authentication Modules (PAM) [3] system in place on most UNIX machines. The AUTH service negotiates authentication technologies and proceeds to engage the client in a mutually agreed challenge/response protocol. When the back-end authentication service is satisfied of the client's identity, the AUTH service relays the results to the client and attaches an identity to the session. After a session has been granted a certain identity, they may be authorized to access restricted services or other resources. The *fived* daemon can also include a mechanism to pass the session's authenticated identity information through to underlying services they connect to.

Figure 2.2: AUTH Command

Finally, depending on the service provider's setup, they may not wish this service be supported until after the user gains a secure channel for their session using the TLS service which is described in the following subsection. In this case, AUTH itself acts as a restricted service until after TLS or another acceptable encryption scheme is invoked.

2.2.1.3 TLS

TLS allows for session encryption. As the name might indicate, the TLS service is a command to the session server to start a TLS handshake. After the client requests this service, the session server and the client immediately engage in a TLS handshake and set up a secure channel. The client should retain the certificate offered by the server for future connections to ensure security. This is similar to how SSH handles key verification and has been moderately more successful than the web-based model for TLS. However, the client should feel free to use other mechanisms, such as the existing TLS PKI, to verify the certificate during first connect. After both the client and the server has completed the TLS handshake, the session continues over an encrypted channel. In

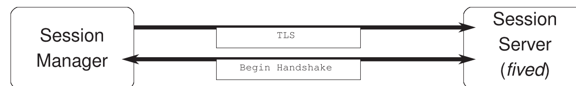


Figure 2.3: TLS Command

addition, the server may choose to authenticate the client on the basis of a client-side certificate they present during the handshake.

Here is an example of a user using the TLS service, then authenticating using AUTH and receiving access to the service *telnet* which they then access over a secure and authenticated session:

≫**LIST**≪
TLS
AUTH
≫**TLS**≪
+ SUCCESS
a TLS handshake takes place and the session continues over a secure channel:
≫**AUTH <up,ext,pubkey>**≪
+ SUCCESS <up>
Enter Username: ≫**researcher**≪
Enter Password: ≫**secret**≪
Authentication as researcher successful
≫**LIST**≪
telnet
research-service

≫**telnet**≪
researcher@researchbox $

2.2.1.4 MULTIPLEX

MULTIPLEX allows more than one service on a session. In the basic protocol, when a client requests a service, the connection is taken over by the service and there is no further interaction with *fived*. Multiplexing allows a user to connect to a session server, encrypt their session, authenticate and then access as many services as they need.

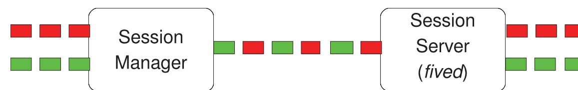


Figure 2.4: MULTIPLEX Command

Figure 2.4 shows a multiplexed session where multiple services are interacting with multiple clients. The client computer runs a session manager that handles the client connections from that machine while the service provider runs services through the *fived* daemon. There is no requirement that the session manager and the clients be on the same machine, nor is there a requirement that the session server and service daemons be on the same machine. This network-transparent interaction allows for organizations to create unified session servers that are the frontend for all of that organization's services. This allows for centralized authentication and also could allow a session server to act as a load balancer for various backend services.

In response to the MULTIPLEX command, *fived* begins the session multiplexing protocol. To multiplex more than one application layer datastream on top of the same reliable bytestream, *fived* uses a series of headers to delineate datastreams. Figure 2.5 shows the header format:

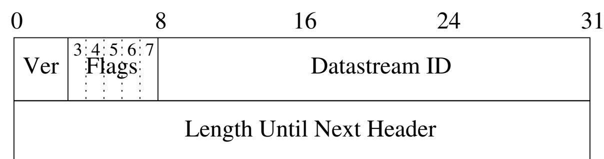


Figure 2.5: MULTIPLEX header format

The header fits in 64 bits, which allows for easy manipulation on most modern processing units. The first 3 bits comprise a version number, the next 5 bits contain flags, the subsequent 24 bits contain the datastream ID which identifies the datastream which follows the header and the final 32 bits is the length, in octets, until the next header. The meaning of the flags are as follows:

- **Bits 3 & 4** – *Reserved for future use.*
- **Bit 5** – *Complete* – This flag is set for a one-sided close in a duplex transport protocol. (As in `shutdown()` in the standard sockets interface.) The side that sends this flag is declaring that they no longer will be *sending* data. The datastream ID is still active, unless or until the other side sends a message with the complete or close flag. The length to next header field must be set to zero when this flag is set. (This header may not proceed data using this datastream ID.)
- **Bit 6** – *Close* – This flag is set when the application using this datastream is no longer willing to communicate. The other side should discontinue use. Any data for this ID will be dropped. The length to next header field must be zero when this flag is set.
- **Bit 7** – *New* – This flag is set by the server when the user asks to use a new service. The datastream ID will be new and identifies data from that service from now on. The datastream ID of zero is reserved for talking to the *fived* daemon.

This multiplexing protocol is sufficient for a user to access multiple services concurrently using their session. Their authentication stays intact and the encryption continues. The session persists until the user closes their connection to the session daemon.

2.2.1.5 HOST

Figure 2.6: HOST Command

HOST enables a session server to provide services for many hosts, virtual or physical. Depending on the host selected, different services can be enabled. When

a client issues the HOST command they provide a hostname or service-group name. Assuming the session's privilege level is sufficient and the name the client requests exists, the daemon issues an affirmative response and associates the session with the requested hostname. This allows the session layer to do virtual hosting at a network level. This allows organizations to centralize sessions into a small set of session servers which act much like load balancers do in existing networks.

2.2.1.6 DETACHABLE

DETACHABLE allows a client to disconnect from a session without destroying its state. If allowed by the server, DETACHABLE is a mechanism to request the server maintain a session's state while a client disconnects from the server for a time. This is almost a network equivalent of the UNIX screen [1] command. The DETACHABLE service provides the user with some sort of secret. This secret could be a cryptographic certificate, a password, ASCII art or any piece of data appropriate for the security requirements of the session. When the client disconnects from the session server, the session's state persists. Data from services which remain open will be queued. The amount of time a session's state is perserved and the amount of traffic it is willing to queue is up to the administrator of the session server.

2.2.1.7 ATTACH

ATTACH allows a client to resume a previously detached session. The user provides the secret issued by a previous invocation of the DETACHABLE service along with the number of bytes they've received since the session began. After verifying the secret, the user will be allowed to resume their session. However, since the user is likely to want to start a TLS session before providing the secret to the ATTACH service to prevent man in the middle attacks, resumed sessions will use this new TLS session, if one exists, instead of resuming an old one. (This also provides a re-key mechanism for long-standing sessions.)

2.2.1.8 Broader Uses of DETACHABLE and ATTACH

It is not required to break a session connection before using ATTACH on a DETACHABLE session. Instead, a user can attach another layer 4 connection to their existing layer 5 session. This allows different quality of service properties or connection bonding. DETACHABLE and ATTACH can also be used on one specific connection, which allows users to gracefully roam networks or even physical machines.

2.2.1.9 PROVEAUTH and GETSIGNKEY

This service provides a lightweight distributed identity system. PROVEAUTH allows a user to use their session to prove their identity to another system. Where AUTH creates a system of authentication for the session layer, PROVEAUTH allows a user to prove that identity elsewhere. This allows users to use an identity from one entity to authenticate with another.

In these types of protocols, there are three parties:

The identity provider (P) This is the entity providing the identity. It holds an authoritative notion of identity for its domain and chooses to grant these identities to users. In *fived* this entity is the session server providing the PROVEAUTH service.

The user (U) This is the end-user of the identity. In our session protocol, this is the user controlling the session client.

The identity consumer (C) This is a separate entity who accepts identities asserted by the identity provider and wishes to ensure that the user has a right to use a particular identity.

The protocol for a user to prove an identity to an identity consumer is as follows:

1. User (U) sends the identity to identity consumer (C) they want to prove is theirs.
2. Consumer (C) responds with a challenge to user (U). The challenge consists of a nonce chosen by C along with the canonical name for C.

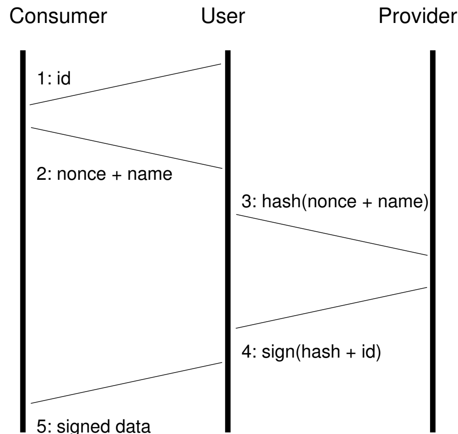


Figure 2.7: PROVEAUTH Protocol

3. User (U) concatenates the nonce and canonical names provided by the consumer (C) and hashes them. User (U) then invokes PROVEAUTH on their authenticated session with the identity provider (P) and provides this hash.
4. Provider (P) concatenates the hash with the identity the user's (U) session is authenticated as and signs the result using an RSA keypair whose public component is known by the consumer (C).
5. User (U) returns this signed data to the identity consumer (C).
6. The identity consumer (C) proves that user (U) has a right to the identity by verifying the signature on the data and ensuring the contents of the signed message matches the identity, nonce and canonical name expected.

This protocol allows for something many other distributed authentication protocols don't: it allows users to use their identities elsewhere without revealing who they pass their identity too. Unlike other major protocols (Facebook Connect, Twitter Auth, etc) where the person who controls your identity has a complete list of where you use it and when, this protocol omits the ability for identity providers to engage in that level of tracking.

GETSIGNKEY is a convenience service which offers the public key used to prove identities in PROVEAUTH. This service provides one mechanism out of many

that identity consumers could receive the public keys for the signing keypairs for the identity providers they wish to support.

2.2.2 Session Initiator

Fived's session initiator has a larger goal of breaking network applications' dependence on the lower layers of the Internet. One of the major bottlenecks in the existing transition in-progress between IPv4 and IPv6 is that applications are required to be aware of IP addresses. This knowledge is necessary for applications even though users mostly specify computers by hostname. Yet, the application itself is responsible for the name resolution. Once it resolves the name, it must pass the correct layer 4 address to the underlying networking APIs.

The session initiator changes this. With it, the session API and session architecture take control earlier. The session initiator performs the initial connection establishment on behalf of the application. This allows applications using the session stack to move beyond the existing APIs focused on addresses and port numbers and simply ask the networking stack for a service. The session initiator needs to know two things: the name of the organization or computer the application would like to communicate with and the name of the service the application would like to access. With that, it does the rest and sets up the session.

Once applications move away from using addresses and port numbers, the underlying architecture of the Internet can evolve without nearly as much hassle. The session initiator will be the only thing that needs to change to allow applications to connect to each other in new ways. Arguably this only moves the problem around, but importantly it moves it to a place better designed for change, future expansion and alteration. The session layer is a more appropriate abstraction and interface for applications on the Internet.

2.3 Performance

Performance is always a critical issue. Users see performance overhead as a cost to almost any security technology. The cost users are willing to accept varies widely, but

it seems fairly clear that the higher the performance cost, the more difficult adoption becomes.

The performance concerns for *fived* lie in two main areas:

- The increased cost of connection establishment with the session layer.
- The increased overhead during a data transfer across the network.

From the perspective of a user, these two things can be measured with two metrics. The first is the amount of time required from beginning a request until the first byte of data is available to the endpoint application. This is commonly referred to as "Time To First Byte" (TTFB) and generally establishes a lower bound on network latency. The second metric is the amount of time it takes for a request to *complete*. This is harder to establish for *fived* since a session layer is generic infrastructure that supports a variety of protocols with a variety of users and uses. There's no firm definition for the end of a request. So the metrics I used were Time to Thousandth Byte and Time to Millionth Byte, which roughly correspond to a small one kilobyte data transfer or a larger one megabyte data transfer across the network.

Experiments were conducted across the Internet using remote endpoints in two different cities. Average round-trip latency between the computers was 28.36 milliseconds with a standard deviation of 3.40 milliseconds. No significant packet loss was measured on the link. The server side computer was connected to the Internet on a university network which routes to the Internet via fiber, similar to most datacenter environments. The client-side computer was connected to the Internet on a lower bandwidth connection which is similar to most residential environments.

Measuring 500 datapoints shows the Layer 5 Time To First Byte is larger than the Layer 4 Time To First Byte, showing the expected performance degradation caused by needing to interact with a session server before being able to start an application protocol. While the difference is highly statistically significant, there is also overlap between the standard deviation of each data set as shown on the graph. Which means many individual uses of the session layer will not be significantly distinguishable from ordinary network jitter.

The story gets better when you look at time to completion. The gap between Layer 4 metrics and Layer 5 metrics narrows as more bytes are transferred across the

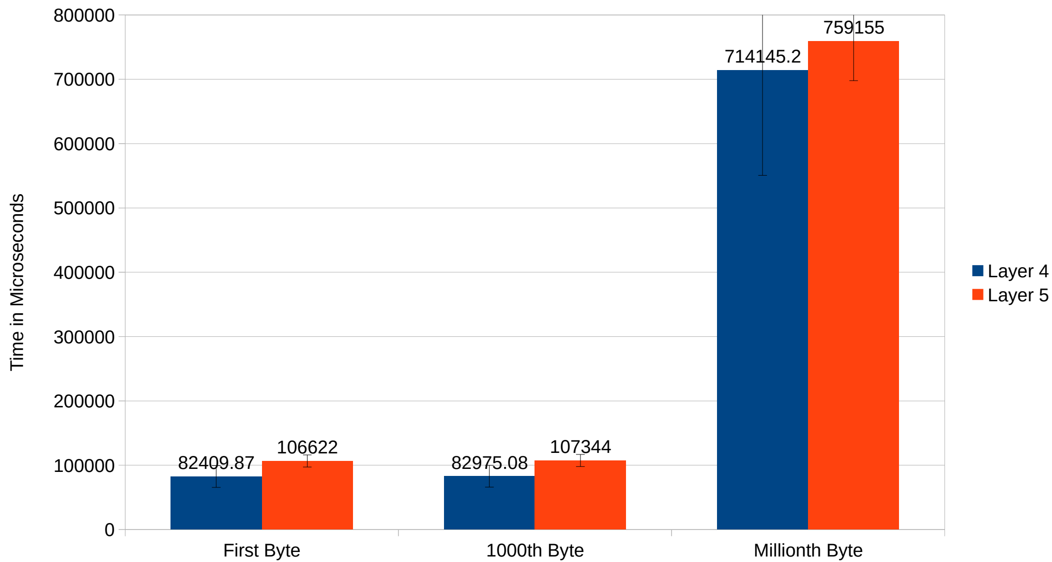


Figure 2.8: Layer 4 vs Layer 5 Performance

connection, which shows the dominating performance impact of the session layer is in the initial establishment of the session. While this doesn't show up as much with a short transfer of a thousand bytes of data, the performance gap at a million bytes of data is substantially lower.

For an implementation of *fived* entirely in userspace with no kernel components and several remaining optimization opportunities, this is not a particularly bad performance picture. It seems likely that the performance of *fived* may be manageable.

Of course, this isn't the whole performance picture of *fived*. The opportunities of a session layer allow us flexibility application protocols otherwise don't have. The data we've gathered so far tells us the story of what happens when you need to get data from a peer across the Internet when you don't have a session established yet and need to set one up before being able to transfer data, but what of the cases where a session exists?

The same experiment from above was repeated with a "warm" connection where a session was already established. In this environment, we show that far from a negative performance impact, the session layer delivers a significant performance *improvement*. Not only during Time To First Byte connection establishment, but all the way through the millionth byte of data transferred. Avoiding TCP slow start appears

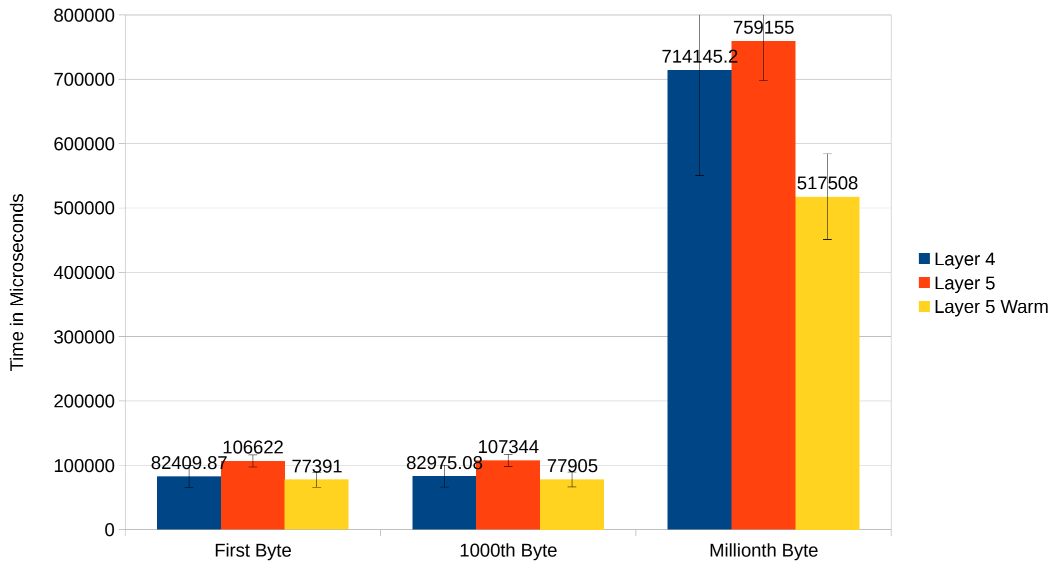


Figure 2.9: Layer 4 vs Layer 5 Performance with Warm Connections

to provide substantial benefit that endures through the transfer.

While this benefit isn't exclusive to *fived* and application protocols have redesigned themselves, sometimes substantially, to employ similar tricks (HTTP 1.1 essentially provided persistent connections and HTTP 2.0 essentially provides multiplexing) the session architecture allows these performance benefits to transparently apply to any protocol running on it. Instead of always imposing a cost, it's quite likely that the session layer can bring performance improvements and optimizations, possibly even tailored to the specific network environment for each computer, without re-writing every application which implements an application protocol.

2.4 Potential Improvements

2.4.1 Integration into the Kernel

Integrating some parts of the *fived* client into the kernel may provide considerable opportunities for performance enhancements resulting from less context switching, less copying between buffers and other clever opportunities that occur in kernelspace with full access to the kernel networking stack. The *fived* server could be similarly

accelerated, though it seems fair to say that while demultiplexing might be suitable for inclusion into the kernel, a good portion of the logic in the *fived* server could remain in userspace where it can be customized and easily changed.

2.4.2 Integration into Hardware

For larger networks and integration of *fived* into switches and routers, it makes sense to develop specialized hardware. A considerable amount of the active work *fived* performs during most connections is reading the multiplexing headers and simply forwarding traffic. This could be very efficiently implemented in hardware.

Fortunately interacting with the *fived* server directly is a rare operation. It seems acceptable to expect that the core routing and forwarding portion use hardware acceleration, while interacting with the session layer (i.e. requesting new services) can be an exceptional operation handled outside of hardware or even by a separate machine. This machine then handles the request using higher level processing power and then hands down a new forwarding path for the accelerated multiplexer and demultiplexer to use when there's a new connection being established.

2.5 Deployment

Deployment needs to be a key concern with any new networking technology. The goal is to ensure that no network or set of users find themselves unable to benefit from *fived*. With the design of *fived*, an end-user can start gaining the benefits of a session layer if any of the following occur: 1) Their operating system vendor incorporates it into the networking environment for the operating system. 2) The network operator deploys session technology on their network. 3) The user runs any application that natively uses *fived*'s session layer. If **any** of these conditions apply, compatibility toolsets will allow most users to gain some of the key protections and benefits of *fived*. In this section, we talk about how *fived* can be successfully deployed in each of these three cases.

2.5.1 Deploying with Unmodified Applications

Naturally, today's applications do not already support *fived.* However, using a shared library preload unmodified applications can be retrofitted to use the session layer via a compatibility shim. The shim could intercept calls made to the networking interface, including name resolutions, sets up a session to the requested destination, and routes traffic through the session. These applications would then be able to transparently take advantage of a user's authentication credentials on a session or any increased access level, session mobility and reconnection features or transparent encryption services supported by the server. In addition, in the case that the application uses an deprecated connection protocol, the shim could convert its API calls into a request to the session initiator. This allows applications to use protocols that didn't exist at the time the application was written to reach services.

In the case of an unmodified server application, no changes are required since the interactions servers have with the *fived* daemon appear no different than any other network connection. However, a user who wishes to *only* expose a service through *fived* will have to reconfigure their server to bind to a location on the machine only *fived* can access. The easiest way to do this is generally to configure servers to bind themselves to localhost or a local socket. It should be noted though, that users are free to expose services via *fived* session layer while still keeping them open to all non-session layer users via traditional means.

With *fived* users are not forced to take an all or nothing approach, the migration to a session layer can happen slowly. *Private* services that aren't intended to be visible to the entire world will generally be the easiest to migrate. Since many private services are offered by organizations to people affiliated with them, it's easier for these organizations to require people install a *fived* client. For public-facing services, it will take time to get to a point where all users have migrated and support for direct layer 4 connections can be disabled on legacy services.

2.5.2 Deploying with Unmodified Computers

Another compatibility option allows a network administrator to session-enable segments of their network without waiting for each of the individual hosts to get their

own native session support. In this scheme, a DNS proxy (or similar technology) can point outgoing name resolutions to a session server. This session server can initiate sessions to the hosts requested. This could allow for the users on the network to access things they wouldn't otherwise be able to reach in the case that the network has an established authenticated session to some other organization, or it could merely ensure traffic traverses a wide area network over an encrypted link. For mobile vehicles, like buses, boats or airplanes, networks could also use the mobility features to maintain connections as the vehicle roams between network points, possibly reconnecting from an IP in a completely different Autonomous System.

A similar system could be built at the network segment perimeter without a cooperating DNS server. If the session client is placed along the route for outbound traffic, it is free to examine the destination of the packets coming from the unmodified host, open a session to that destination and tunnel the traffic over the session layer. Whether or not DNS spoofing provides network administrators and users with a more desirable solution is an open question.

Chapter 3

Changing the Computer Organization

The realization that modern architectures lack sufficient security features is not a new one. Many grand visions of "trusted computing" remain unfilled. After millions of dollars of research money, huge expenditures on the part of industry and rarely seen levels of cooperation between hardware vendors, users remain without substantial security features in their systems. Those features that do exist remain unused and few users urge progress. Even the strongest advocates of trusted computing have quietly ratcheted down their expectations and the current proposed uses for the TPM [72], a chip that was supposed to bring about revolutionary security benefits to modern computing, represent a significant departure from the original vision of the project. Despite a rapidly increasing need for security, trusted computing systems remain unimplemented, unrealized or without adoption.

Over the last few years, I designed an alternative system called *LockBox*. While it is not possible to fully characterize an unimplemented system, the design represents what I felt at the time to be the next generation of trusted computing technologies. And indeed, as the last generation of trusted computing hardware has faded away, Intel has put Software Guard Extensions (SGX) [4] on their product roadmap. Intel's SGX technology presents a surprisingly similar design to *LockBox* and accomplishes many of the same goals. The design work we've accomplished, along with the preliminary data we've gathered about how a system like *LockBox* might work, is perhaps even more critical now that hardware with a very similar design is slated to hit the mass market within a few years.

First I explain the design tradeoffs we made in designing *LockBox* and highlight where our system made different choices than Intel appears to be making with SGX. Second, I describe the technical design of *LockBox* in detail. Finally, I share the design ideas and data we gathered over our time exploring these types of systems and highlight a few areas that may be increasingly relevant as we see these systems develop a broader ecosystem and head towards widespread adoption.

3.1 Goals

A key issue which prevented adoption of the last wave of trusted computing platforms focused on user control [5]. Almost all previous trusted computing platforms were proposed as a way to secure content from users. While this may be a valid goal in some cases, the purpose of these systems is to remove control from the end-user of the device, which did not make them particularly popular. One of the key design principles in *LockBox* is that the end-user is given tools to control the security of their computer. In a reality where user-error is a large contributor to producing many security issues that people face, trusting a user to make security decisions is an unfortunately controversial approach. However, the only way to provide security features that enable users rather than restrict them involves putting security decisions in their hands. The merits of a security system that is controlled by someone other than the end-user is irrelevant if the end-user rejects that system. *LockBox*'s approach provides a feasible path to enabling security features for end-users.

LockBox allows a user to tell the system which applications they trust. *LockBox* provides a mechanism for the user to provide sensitive data to the system and ensure that only the application they trust with that data has access to it. *LockBox* ensures that this data remains secure if the trusted application is correctly programmed and it ensures this security even if the management software between *LockBox* and the trusted application is malicious.

LockBox provides these assurances by embedding features for *Trusted Loading*, *Trusted Memory*, *Trusted Runtime* and *Trusted Channels* into the architecture. The *Trusted Loading* features ensure that the user can identify and correctly load trusted applications. The features for *Trusted Memory* ensure that only the trusted applica-

tion can access the secrets assigned to it, and the *Trusted Runtime* features provide the trusted application with protection from malicious privileged code. Finally, the *Trusted Channels* features provide the user with a way to exchange data with trusted applications.

LockBox enables users to protect themselves from bugs that result in completely malicious actions on the part their operating system or even their hypervisors. This is a critical distinction as many previous trusted platform proposals require an entire trusted software stack where all management software is required to be correct and bug-free. Yet, operating system bugs are all too common [34, 52] and as hypervisors get more complex, we are likely to see a similar progression. *LockBox* provides a way for data to remain secure even when these systems fail.

The design of *LockBox* requires defining a mechanism for the security system to prevent untrusted management software from compromising sensitive data while still allowing the management software to do its job. In general, this is accomplished based on a *request/verify* procedure where the trusted application makes requests to the operating system, but *LockBox* ensures a trusted entity can verify that the request was properly performed and the management software is properly functioning. Since *LockBox* preserves the management software's abilities to manage the system, malicious management software is allowed to manage processes and even kill a trusted application. However, even if the application is killed *LockBox* ensures that sensitive data cannot be retrieved. Reclaiming protected memory can only occur after that memory has been zerofilled.

These features move beyond the old definition of a trusted computing platform and incorporate real features that provide security benefits in computing systems. Systems that implement these features are necessary to enable an increased range of activities on computing systems. Current environments don't achieve the levels of security required for the full range of tasks users would like to do on their computers. Systems like *LockBox* may be necessary to achieve those goals.

3.2 Related Work

A large body of work has developed around trusted computing. However, the most visible and well known type of trusted computing platform remains something very different than what we discuss here. The Trusted Computing Group [73, 72] is a widespread industry effort supported by virtually every major technology company and produces a shipping product. For many years, their research dominated the discussion around trusted computing. There are also academic systems, like Terra [30], with similar goals, though Terra is implemented entirely in a hypervisor. Like the *LockBox* design, these systems provide a mechanism for memory protection called sealed storage and, in the case of the TCG's system, involve hardware modifications. Yet, unlike *LockBox*, both Terra and the TCG's system are designed to produce a full trusted software stack in which software at all levels of the machine is trusted. These types of trusted platforms are very different from *LockBox* because of their reliance on this trusted software stack.

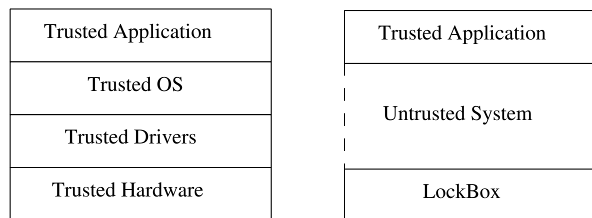


Figure 3.1: Full Trusted Stack vs. LockBox Design

Another set of systems, XOM [43, 44], AEGIS [69] and others [7, 20] are all systems which also embed security features into the hardware. XOM and AEGIS are of particular interest as they do not require a trusted software stack. Also in this category are the hypervisor-based systems Overshadow [16] and HARES [71]. While all these systems wrestled with many of the same issues present in the *LockBox* design, all of them chose to implement memory protection using an encryption layer. *LockBox* takes a different approach and eschews the use of encryption algorithms. Under *LockBox*, if one piece of code isn't authorized to see the contents of a page, then that piece of code isn't allowed to see the data in any form. The hardware returns zeros on reads and discards writes. This alteration fundamentally changes the relationship between the operating system and the rest of the computer organization.

This change requires new solutions in some areas that previous work does not address. Swapping, for instance, becomes an issue when the operating system isn't allowed any way to view protected memory. The reason *LockBox* is different from all other previous work in this respect is that unfortunately, allowing an operating system even an encrypted view of memory provides more information than it might seem. While the operating system may not be able to decode the exact contents, the OS will be able to track which data changes at which times and how much, which is an overlooked information leak in these systems. Not only is AES's 128-bit blocksize an issue, but memory's random access nature precludes the use of chaining modes without substantial performance degradation. The information gained by watching which portions of memory changed can be quite detailed. In Section 3.5, an attack against these types of systems is outlined and we describe the suite of tools we developed to analyze these weaknesses. Further, since *LockBox* doesn't require the use of encryption to implement its core security features it not only has the potential to perform better, but the system does not require specific cryptographic algorithms to be set in stone.

In most systems, the end-user has little control over which applications are allowed to use these system's security features. Therefore, the user is often placed in a situation where an undesired application uses the security infrastructure not just to prevent them from, say, copying protected content, but also to create powerful rootkits [48]. Further, this lack of control by the end-user has been a large inhibitor to the adoption of trusted platforms [5]. *LockBox* addresses this issue by placing the user directly in control of the authorization and attestation processes. One of the research challenges in this proposal is defining how security architectures can directly interact with users in a sensible way. In Section 3.4 we present data from a user study that provides insight on how some of these interactions can be made more successful. *LockBox* stands alone in involving the user directly in the attestation and authorization workflow of a trusted platform. This is the only viable way for trusted computing to achieve acceptance in the marketplace and even by the full security community, parts of which have grave concerns regarding current systems.

Finally all of these systems specify either a trusted hypervisor or trusted hard-

ware[1]. *LockBox* on the other hand, was designed to be able to run as either a trusted hardware design or as a pure hypervisor framework. This allows legacy machines to use *LockBox* with a microhypervisor while allowing for a smooth transition to a hardware implementation at a later time.

Some other pieces of the field are relevant to the development of these types of systems. Single Address Space Operating Systems [75, 15, 40] provide the basis for *LockBox's* SLB structure. Hypervisor monitoring systems like [62, 25] contribute some implementation level tips and tricks to make closely meshed page tables in hypervisor systems perform well. Work on hypervisor nesting with bluepill attacks [54] on Xen [8] as well as KVM's [39] production level implementation of hypervisor nesting have paved the way for hypervisors to emulate hardware features without eliminating the user's ability to run a more feature-rich hypervisor that accomplishes other objectives. In addition, various microhypervisor systems like Bitvisor [64] and SecVisor [59] have provided interesting insights into how security features should be incorporated into small hypervisors. Systems like vTPM [11] demonstrate the feasibility of simulating security hardware inside virtual machine monitors.

Multics's memory protection rings [55, 57] were an early form of memory protection and an important comparison point for future designs. Other approaches to system security include operating system hooks to produce fine grained security policy [65, 46] and better tools [18, 60, 35] to help software eliminate security defects and move closer towards correctness. These mechanisms will continue to be an important line of defense in the security of the overall system. Systems like *LockBox* complement much of the existing work in the field of security and provide applications with a last additional line of defense when these other approaches fail.

3.3 Technical Design

The *LockBox* design outlines a collection of architectural features to improve system security. Targeted towards implementation in either a nesting hypervisor or an FPGA, *Lockbox* is designed to examine how end-users could be provided additional

[1]XOM specifies that part of the architecture features could be implemented in a hypervisor, but acknowledges a few important elements would still require specialized hardware.

hardware security features almost invisibly. The benefit of a nesting hypervisor is that a user can still run their own hypervisor on top of the system. This means users can still obtain the benefits of a feature-rich hypervisor without expanding the Trusted Code Base of *LockBox*.

In general, *LockBox* was designed to ensure that once a trusted application is provided with sensitive data, that data remains secure so long as the trusted application was correctly programmed and *LockBox*'s features are correctly implemented. *LockBox*'s features are designed to ensure that users will be made aware of whether or not their secrets are being entrusted to applications running in a secure and trusted context. Even in the case that the trusted application contains bugs, an attacker no longer receives complete access upon compromising intermediate software, but is forced to break both the management software *and* all of the user's trusted applications protecting the data they desire. This substantially raises the bar required to steal sensitive data on a computer.

The design of *LockBox* revolves around four different sets of features:

Trusted Loading These features define the concept of an application and an application instance, and enables LockBox to identify an application instance when it makes a request to LockBox. The architecture grants trusted contexts and trusted identifiers to only the applications a user identifies as one they trust.

Trusted Memory These features protect memory from unauthorized users, including the operating system and other system level software. These features ensure that code running in a trusted context can make use of protected memory which cannot be read or written by other code running on the machine.

Trusted Runtime These features isolate code running in a trusted context from interference from supervisors and/or hypervisors running at a greater privilege level than the application, but at a lower privilege level than *LockBox*.

Trusted Channels These features allow code running in a trusted context to receive input from devices in a manner which identifies the trusted recipient to the user and ensures that this input cannot be read by any supervisors and/or hypervisors running at a greater privilege level.

3.3.1 Trusted Loading

The *Trusted Loading* features of the *LockBox* design provide the notion of identity and allow the architecture to maintain the concept of an application. This allows low-level components to identify applications and provides the basis on which access determination can be made.

Determining which applications a user trusts is another fundamental issue these features address. A trusted platform must be careful about the code it allows to run within a trusted context. If a trusted platform allows all code to run in trusted contexts, malware and other types of undesirable software can use the security features of the architecture to hide data from the end-user. On the other hand, if a trusted platform doesn't provide a mechanism for the user to allow code to gain access to the security features, then these security features are hardly useful. *LockBox* contains mechanisms to create secure trusted contexts only for those applications that the end-user authorizes.

3.3.1.1 User Access Device

Allowing a user to specify exactly which applications they trust requires the trusted platform design to directly interact with the user to determine their list of trusted applications. Since users tend to have trouble making trust determinations based on a set of hexadecimal numbers, a user interface is critical towards allowing the end-user to comfortably interact with the security system. A human-readable identifier (typically a unicode string) must be created that the end-user will recognize when prompted to input data destined for a trusted application.

Fortunately, an existing metaphor can be reused. People are comfortable with carrying keys to open door locks, car locks and other types of physical security mechanisms. *LockBox* is designed to interface with end-users using a device that they would use just like a key; simply inserting the device into a special designated socket. The key provides digital storage populated with information read directly by *LockBox*. Typically, writing to the device should require the user to physically release a hardware interlock on the key itself once it's inserted into a machine. Users program their keys using a computer or device they trust.

The user's key contains the following information for each application they trust:

A human-readable identifier This is the human readable name the architecture will use to identify the trusted code to the end-user. It should be chosen by the end-user.

At least one of the following machine-readable identifiers:

- **A cryptographic hash** On architectures which support a hashing mechanism in hardware, users can simply store a cryptographic hash that the loaded executable must match.
- **A full executable image** An exact image of what the loaded executable must look like. The hardware compares this against memory contents to check for a match.

While the storage itself must be capable of being read directly by the architecture, the data on the key isn't secret. It simply contains a list of human-readable identifiers that correspond to a list of machine-readable identifiers.

The simple case for the User Access Device is that it is a simple storage device with a hardware interlock which is manually programmed by a user on a trusted machine. However, more complex possibilities exist. If the *LockBox* design were widely deployed it wouldn't be unreasonable to expect that stores might sell User Access Devices pre-populated with hashes for all the most popular trusted programs. In a corporate environment, an employee's name badge could serve wirelessly as their User Access Device. The device could be updated each morning as it interacts with the building's security system when the employee enters; signed updates would ensure security and the employer could transparently provide their employees with consistently up to date User Access Devices data. *LockBox*'s design can support much more complex environments than covered here.

3.3.1.2 Creating Trusted Contexts

While *LockBox*'s design does not trust privileged code to maintain the security properties of an application, it does trust privileged code to manage a machine's

resources. Creating a new trusted context is something that can only be done by privileged code. However, in order to assign the human-readable identifier and create the trusted context the architecture *checks* that the operating system correctly loads the program into protected memory and that the program matches the user's machine-readable identifier. If *LockBox* successfully verifies that the privileged code has correctly loaded a trusted application, a new trusted context is created. Otherwise, the check fails and the trusted context is not granted. This trust but verify model allows operating systems and other magnagement software to serve in its traditional resource allocation and management roles without allowing it to violate security constraints.

3.3.2 Trusted Data

From the moment a new trusted context is created, new protected memory is allocated for the code that runs the context. The ability to utilize protected memory is crucial for trusted applications. While this memory is originally managed by privileged code, once it is actually allocated to a trusted context, it cannot be read or written by any other code on the machine, including the operating system. This is a change to the way memory protection works in current architectures. This set of features provides the *Trusted Data* part of the design.

3.3.2.1 Security Lookaside Buffer

Code inside a trusted context does not have the same memory model as code outside a trusted context. Within a trusted context, an architecture-defined portion of the address space is reserved for protected memory access. Access to addresses within this range does not use the normal page table or Translation Lookaside Buffer (TLB) mechanism for virtual memory but instead uses a component called the *Security Lookaside Buffer* (SLB), which is a simple variant of a standard TLB. The SLB is backed by a separate set of memory security tables which determine the security properties of memory within this address range as well as the mapping of virtual addresses to physical pages.

This separate set of page tables is necessary to prevent a wide range of attacks that occur when privileged code is allowed to modify standard page tables for protected

memory. Instead, when allocating new protected pages, privileged code must specify the identifier of the trusted context which will own the page and an explicit mapping between the protected virtual address space and a physical frame. Once *LockBox* accepts this allocation it writes the new mapping into a private storage location. Once this mapping is written, the memory arbiter begins to enforce the stipulation that the newly allocated physical frame may not be written by anything except a processor core running in a trusted context. Memory remains protected until either an entire trusted context and all its associated data is unprotected, in which case a trusted I/O controller overwrites the entire space, or the trusted context voluntarily releases the page using the page release mechanism described below.

The SLB can be implemented as a separate component or as extensions to the TLB. In either case, it must be managed by hardware and contain the following information:

- A mapping of the virtual page number to a physical frame number
- The identifier for the trusted context which owns the page
- A valid bit

In order to keep the complexity requirements to a minimum, the SLB simply uses the same set of tables for all code running in a trusted context. Since the SLB does not allow write and read operations from one trusted context to complete or return valid data when issued on memory which belongs to another trusted context, there are no security issues with having a shared mapping. Thanks to the large 64-bit address space found in modern processors there is plenty of room to allow for partitioning of the address space.

This maps particularly well to hardware, but can also be implemented easily in a hypervisor. In a hypervisor prototype these features can be implemented using page tables. The trusted I/O controller can be virtual, just a piece of software. This would essentially allow the operating system to use the virtual I/O controller as a hypercall interface to interact with *LockBox*. The SLB, while an important part of the hardware design, can be entirely implemented using the TLB in the hypervisor based design.

3.3.3 Trusted Runtime

The set of features in the *Trusted Runtime* section of the *LockBox* design ensures that code in a trusted context cannot simply be subverted by the management software on the machine. This set of features allows the upper portion of a software stack to run on top of untrusted management software. To keep this trusted code running inside a trusted context secure, the *LockBox* design includes several changes. First, secure data inside registers cannot be exposed on a context switch. Second, as shown by geometry attacks [61], arbitrary transfer of control into a protected code page cannot be allowed as it is equivalent to allowing arbitrary code execution from that same page. Finally, to enable swapping types of operations there needs to be a sensible mechanism to allow privileged code to deallocate protected pages and flush a representation of them to disk.

3.3.3.1 Program Status Page

In the *LockBox* design, *Trusted Runtime* features are enabled with the help of a preallocated page within every executable which gets loaded into a trusted context. This page is called the *Program Status Page* (PSP) and its address serves as an identifier for the trusted context. The page contains the following information in a layout defined by the individual implementation:

Register Flush Set Pointer This is a pointer to an area that will store the working set of registers when a context switch occurs. It is loaded by the architecture on entry to the context.

Restore Lock This field contains space for a lock. This lock is set during the restore handler to avert timing issues and properly deal with concurrency.

Requested Program Counter Contains a copy of what the Program Counter (PC) was before being overridden when the PC (re)entered the trusted context.

Page Release Address Contains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid. This area is initialized to an invalid address.

Swap Handler Address Contains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Interrupt Handler Constains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Address Contains space for a pointer. When set to an address that is not inside the protected address space this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Length Contains an integer set by a trusted I/O controller when a trusted channel has been closed.

Trusted Device Type Contains an integer set by a trusted I/O controller when a trusted channel has been opened. The value of zero is invalid.

Default Register Flush Set Space for flushing one set of registers. When the PSP is initially created, the *Register Flush Set Pointer* field points to this area.

These fields are used by the trusted application and *LockBox* to enable the following features:

3.3.3.2 Protecting Registers

In normal operation, an operating system can pre-empt a process at any point and view its architectural state. Unfortunately this is no longer acceptable when a security system is in place which allows a program to place data in its architectural state which privileged code is not permitted to see. Therefore, any time the processor forces a transfer of control to a location outside the trusted context, the registers are flushed to area specified by the trusted context's *register flush set pointer* and cleared.

3.3.3.3 Preventing Arbitrary Jumps

Not only is there a need to prevent arbitrary jumps back into code running in a trusted context for security reasons, but there is also a need to restore the sets of

registers which get purged every time the control flow unexpectedly leaves the trusted context. For both issues, the solution is the same. Upon re-entering code that is part of a trusted context, the processor should override the PC and enter the code at the beginning. The old PC value gets placed in the program status page in the *Requested Program Counter* field. In a hypervisor version, this can be implemented using the NX bit on the page tables.

This means at the beginning of every program designed to run in trusted mode there should be either an instruction which transfers control to a restore handler or the restore handler itself. This handler can take care of such tasks as restoring a program's register set from the program status page, resuming execution at the point the program was interrupted or checking the *Requested Program Counter* field to determine if control flow should go to a different location. If the requested program counter is set to a value permitted by the application, the application transfers control to that location.[2]

3.3.3.4 Concurrency

The restore handler not only checks for a valid entry point and restores the registers, but plays a critical role in ensuring that *LockBox* works correctly with multiple threads in either a unicore or multicore system. The *Restore Lock* field in the PSP is the main tool the handler and architecture use to coordinate. When the architecture transfers control to a protected memory region, it acquires this lock. If the lock cannot be acquired, then the architecture may either retry, or return control to the operating system. This allows the restore handler to ensure that each execution context receives a unique location to flush its registers.

To take care of control transfers during the time the restore handler is running, the architecture does not enable flushing registers to protected memory until after the *restore lock* is released. Instead of flushing the register set, the architecture releases the *Restore Lock* and transfers control back to the OS. Normally, the *Restore Lock* is released at the end of the restore handler with an atomic write instruction. This write signals to the architecture that the context is again prepared to receive execution contexts and the current execution context is ready to writeback its registers to the area

[2]Implementations with concerns about the performance of this section of the architecture could implement portions of it in hardware or provide instructions which will accelerate these operations.

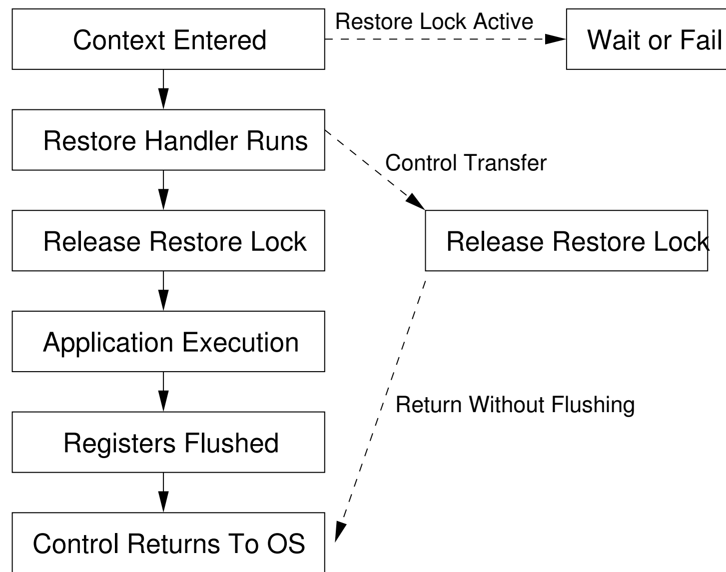


Figure 3.2: Context Switch Handling

specified by the *Register Flush Set Pointer*.

While the *Restore Lock* serializes execution in the restore handler, it does not prevent applications from taking advantage of multiple execution contexts. (i.e. multiple threads) The operating system is free to pass a thread ID to the restore handler in a register, (or any other mechanism defined by the OS ABI) which the restore handler can use to resume the appropriate thread. The architecture reads the value of the *Register Flush Set Pointer* field *on entering the context* to determine where the thread's register set will be flushed. The restore handler sets the pointer for where the *next thread* will flush its registers and thus can ensure that each thread within the trusted application flushes their registers to a unique spot. The restore handler should also record the location of the *Register Flush Set Pointer* for the current thread in a table so that later invocations of the restore handler knows where to find the thread's register flush set to properly restore execution after a context switch occurs. Lastly, the restore handler zeros the PC from the register flush set of the current thread so future invocations of the restore handler can detect whether or not the current thread has finished. (When the current thread finishes, the non-zero value in the PC will get flushed to the flush set.)

In the case of a single threaded application, the locking mechanisms are still required to prevent overwriting valid register flush sets from context switches inside the restore handler, but the remaining issues are simplified. The *Register Flush Set Pointer* can remain at its default value, pointing to the default *Default Register Flush Set* and the restore handler simply returns when the operating system tries to enter the protected context while an existing thread is running somewhere else in the context.

3.3.3.5 Page Release Mechanism

LockBox's design contains a page release mechanism which allows a trusted context to release a page of protected memory from its control. This is an important portion of the architecture; without it, an operating system would be unable to swap protected memory. So this mechanism allows a trusted context to release a page from security protections. If the context does so, it is responsible for either purging it of secrets or replacing the data with an encrypted version which can be handled by untrusted code. Existing work with encrypted swap shows low overhead and hardware acceleration instructions are now present in almost all new chips produced by major vendors. In this environment, the performance of "software encryption" for swap is likely negligible.

To perform a swap operation, the operating system sends a signal to the process which lets the process know that the operating system had determined that the page at a specified address should be swapped out. The program can either refuse to unprotect the page and risk that the operating system will then decide to kill the entire process by requesting the architecture destroy the entire trusted context, or it can comply. If the program chooses to comply, it will likely want to replace the contents of the page with an encrypted version. The program will then place the address in the *Page Release Address* field of the PSP and transfer control back to the operating system, which can then release protections on the page, and flush it to disk.

If the trusted context ever tries to read or write to a protected address for which no mapping exists, the processor jumps to the *Swap Handler Address* listed in the PSP for the trusted context. The handler should request the operating system swap the page back in and generate an access violation if the OS does not have the

page. A well-written handler will also want to verify that the contents are correct and decrypt them if needed. Various cryptographic mechanisms can be used to verify integrity, including cryptographic signatures and cryptographic hashes. If the PSP does not contain a valid handler address, the processor traps to the operating system just as if it was unable to load a TLB entry for that address.

3.3.3.6 Co-operative Swapping

Trusted computing architectures such as [16, 44] allow operating systems to swap protected pages without much issue through the use of cryptography. These architectures provide two views of memory, one encrypted and one decrypted. Applications within a certain trust domain are given a decrypted view of memory while everything outside it, including the operating system, is given an encrypted view. This works well for swapping as the Operating System can simply flush the encrypted copies of the page to disk and restore the encrypted version as needed. In *LockBox*'s design however, the lack of a PKI or any sort of hardware encryption prevents us from following this model. Instead, *LockBox* relies on a *co-operative swapping* mechanism.

To set up co-operative swapping, an application performs the following steps:

1. The application produces a key based on trusted information requested from the user through a trusted channel.

2. The application uses *mlock()* or a similar call to request that the operating system wire down the pages that contain: 1) The code used to handle swap requests 2) The code used to encrypt and decrypt pages 3) The page where the key is stored.

3. The application installs a handler for swap-out requests with the operating system.[3]

4. The application places the address of the handler for swap-in requests in the *Swap Handler Address* field of the PSP.

[3] Recall that the *Requested Program Counter* field allows for applications to permit code from outside the trusted context to jump to certain well-defined addresses within the protected address space.

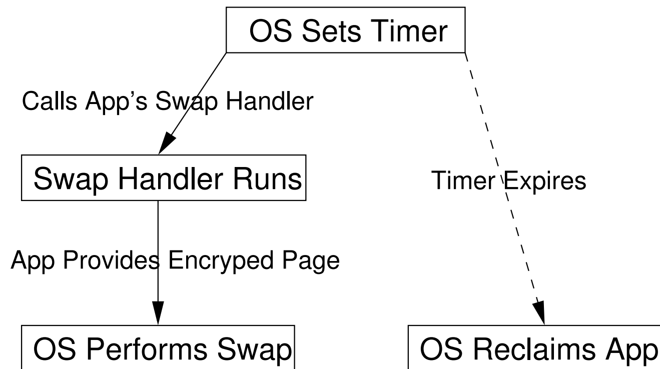


Figure 3.3: Co-operative Swapping

Once these operations are performed, the application continues running as normal; if at any time the OS wants to swap out a page of protected memory, the program uses the *Page Release Mechanism* described in previous section.

3.3.4 Trusted Channels

Trusted Channels are needed for a trusted application to communicate with the user and obtain secret information. The initial design work around *LockBox* largely focused on trusted input. This allows for users to provide secret data to trusted applications without fear of that data being intercepted. The minimum trusted output necessary to implement *LockBox* is the application name. More complex outputs are only needed if applications need to display confidential data. For some common use-cases, (i.e. password entry) securing only input is reasonable and allows us to implement real security features without the complexity of outbound trusted channels.

To create a trusted channel, the application requests one from the operating system. Assuming the operating system decides to comply with this request, the OS sets a trusted I/O controller to handle interrupts from the device with which the channel was requested. The trusted controller writes the type of the device into the *Trusted Device Type* field of the trusted context's program status page. From this point, a trusted channel is now open.

The trusted DMA engine then provides the device with the human-readable name associated with the trusted context. The device can then display this information

to the user. This information allows the user to know two things: 1) Their information will be going to a trusted application and will be stored in secure memory 2) Their information will be going to the trusted application they see displayed by the device. This allows them to determine whether or not to use the device to reveal secrets to their trusted applications. If there is no name displayed, the user will know that the hardware is not securing any of the secrets they provide.

When data comes in from the device on the other side of the trusted channel, the trusted controller places it at a protected page pointed to by the *Trusted Data Address* field in the trusted context's PSP. It then clears the trusted device type field, writes the length of the data written to the page to the *Trusted Data Length* field and closes the channel. Outbound channels will be implemented using a very similar mechanism, but in reverse.

3.3.5 Designs for Trusted Networking

The *LockBox* design opens the possibility of protected contexts where a trusted application can maintain the confidentiality of its data even if the management software running between *LockBox* and the application is malicious. Using these channels to enable trusted applications to securely move data on to and off of a network from their trusted context opens many possibilities. In this section, various models for interacting with a network using trusted contexts are examined.

Perhaps the most important property needed is access to the network which does not require passing sensitive data to management software running between a *LockBox* system and the end application. Existing systems require applications pass network requests to the operating system which multiplexes the network interface with its own networking driver. In the *LockBox* design, trusted channels provide direct channel to a networking interface. A trusted application can pass data directly with a virtualized network interface and does not rely on the operating system to multiplex the network interface.

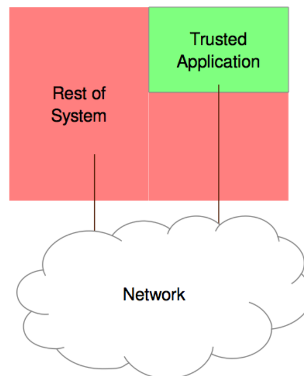
The second security feature important to a trusted context is maintaining a notion of identity on the network. Being able to securely communicate on the network is of limited value if the network can't disambiguate data from a certain trusted context

with from any other data flowing out the same interface. Enabling a trusted context to maintain an identity on the network is a key requirement for trusted networking.

There are two different types of identity we think are critical in a modern trusted system. The first is identity on the local network. This identity allows the networking infrastructure to disambiguate a trusted context's traffic from other traffic on the local network. The second is identity on the global Internet. This identity allows traffic from a trusted context to be distinguished by other applications running on other computers anywhere on the Internet. Both types of identity are important to enable a broad set of applications to use *LockBox*-like trusted channels.

The trusted networking needs for *LockBox*-like systems fall into three distinct modes. These modes can be combined or used separately depending on the security needs of the application. Each mode requires a different set of storage requirements. Each is stated, including whether or not the data stored has confidentiality requirements.

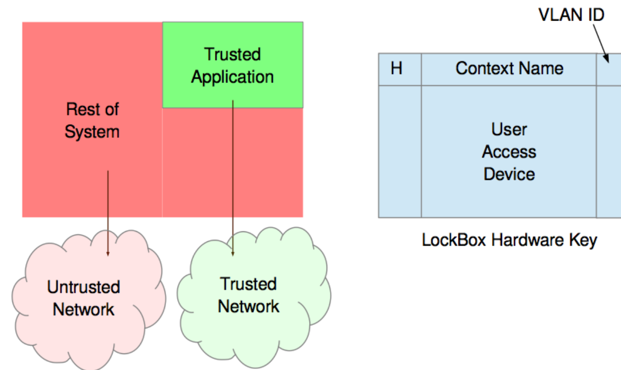
3.3.5.1 Local Equivalent



The first mode focuses on a baseline level of protection that allows a trusted context to communicate on the network without interference or snooping from the Operating System. Single Root I/O Virtualization (SR-IOV) can be used to multiplex a series of virtual interfaces on one network card out to trusted contexts and the host with minimal performance overhead. These interfaces can all be managed independently of each other and traffic sent and received by one interface can be separated from the others in the software stack. If the networking infrastructure and physical links are

trusted, this alone can be enough to ensure some level of security for trusted applications which wish to interface on the local network. If they are not, existing network security technologies can be deployed.

3.3.5.2 Local Network Identity

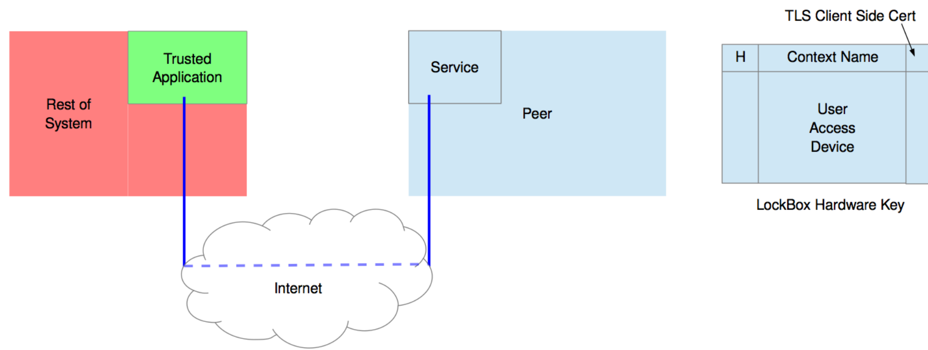


Providing an identity for trusted contexts on a network is a step beyond providing Local Equivalence. Identity information can be present in *LockBox*'s *User Access Device*, the piece of *LockBox* that contains human and machine readable identifiers for each trusted context. The user can include a port and VLAN tag and/or MAC address in the *User Access Device* for each trusted context the user grants a local network identity. Like other data held in the *User Access Device*, the security properties of the system rely on the data's integrity, but not its confidentiality. LockBox can then partition traffic from trusted contexts onto a separate physical trusted port (or several trusted ports, depending on the number of physical ports available on the network interface and the user's desire for separation) and tag traffic from trusted contexts with the VLAN tag and/or MAC address specified in the *User Access Device*.

Many modern network cards have the ability to restrict virtual interfaces so use of a specific VLAN tag or MAC source address can be specified before yielding control over the virtual interface [37]. Cards also include functionality to detect malicious drivers, which allows *LockBox* to prevent other virtual network interfaces running on the same trusted physical port from using an identifier which was not assigned. These features are embedded within the hardware with minimal performance overhead.

These tags can be used by the hardware in the underlying networking infrastructure to make access control decisions. Since *LockBox* can dedicate a physical port exclusively to traffic from trusted contexts, the networking hardware can know that any traffic on that port with the appropriate VLAN or MAC source tags came from the trusted context associated with those tags. Traffic on other ports can remain unrestricted.

3.3.5.3 Internet-wide Identity



VLAN tags and MAC source addresses only provide an identity to the local network infrastructure. While this provides an identity on a local network, many interesting security applications span administrative domains. They need identities that reach beyond the local network. This can be achieved with TLS client certificates [24] and like before, can be stored in *LockBox*'s *User Access Device*. Unlike other identity information stored in the *User Access Device* however, the confidentiality of this data does impact the security constraints offered by the system. Only the trusted context assigned to the TLS client certificate should be able to read the private key material in the certificate

LockBox ensures that once a trusted context is properly initialized and verified, the client certificate assigned to the context can be loaded into its protected memory and remain confidential. The context can then use this certificate to open TLS connections across the Internet and verify its identity using its certificate. Recently, a large amount of work has been put in to reduce the overhead of TLS as deployment of HTTPS has

increased in several major web-based products. (Gmail, facebook and twitter are notable examples.) This overhead reduction combines hardware features like AES-NI [36], a series of hardware encryption acceleration instructions available on recent processors, with a range of software optimizations. Use of TLS in this context provides a high performance and simple way to provide identities across the Internet.

3.4 Designing Interaction with Users

End-users frequently disregard, respond erratically or don't even notice the interfaces that are supposed to help them protect their machine. Here we examine hardware interfaces that provide end-users with an auxiliary security screen which displays information from a hardware security system. We are interested in finding out how users react to such a screen and determine whether or not such an interface allows user to make good security decisions.

I designed an experiment where users were prompted to log in to a series of login forms. The screen displayed various iterations of text and we were able to test user's error rates as well as response times to various security displays. I observed both the timing information and the error rates of each screen. With statistical analysis, we draw several conclusions from the data. These supported some of our interface design assumptions, provide insight into phishing type attacks on these types of interfaces and suggest how the hardware should respond when there is no trusted channel present. These conclusions validated portions of the *LockBox* design as well as suggest possible tweaks for further accuracy in the future.

3.4.1 Experimental Design

There are two very important metrics that determine success for a security system like this:

Accuracy The user's ability to come to the correct conclusion.

Cognitive Overhead How much thinking the user is required to do to come to a conclusion.

While many, might consider the first metric to be most important, the second can be just as, or more, important. The amount of cognitive overhead a user experiences *determines their willingness to use the system at all.* Since the accuracy rate of a system that a user doesn't use is zero, it could be necessary to trade accuracy for lower cognitive overhead. That said, this data does not support the existence of a tradeoff between these two variables in this particular interface. Our results are consistent with the hypothesis that interfaces with lower cognitive overhead are also more accurate.

Accuracy metrics were gathered by recording whether a user reached the correct decision. Tracking cognitive overhead was more difficult. Since there no neuroscientist or MRI machine was present in our lab, I recorded time between responses. This seems like a reasonable metric for cognitive overhead, but simply observing how quickly someone reacts is an indirect measure.

3.4.1.1 Subjects

20 subjects participated in the study. They were recruited from the University of California, Santa Cruz campus and Silicon Valley. The academic participants from UCSC's Computer Science department consisted of one professor, one undergraduate and five graduate students. In addition, two undergraduates and four graduate students participated from other departments at UCSC. Another was studying an unrelated field at Stanford. The remaining six participants worked full time for various Silicon Valley tech companies.

The subjects ranged from 20 to 50 years of age. All were familiar with technology, but not all had chosen it as their vocation. Most were students. In general, the subjects were both young and very familiar with computers.

3.4.1.2 Pilot Study

The first six participants were comprised the pilot subjects for the project. When it was found the pilot study was yielding valid data and the methodology appeared to be sufficient, we incorporated data from these subjects into our main study. The methodology did not change after the pilot study. At the conclusion, we verified that the pilot study participants did not produce significantly different results than the remaining

users.

3.4.1.3 Test Procedure

The test began by providing the subject with a series of instructions. Included in these instructions were information on the test, the methodology and the format. Users were given a username and password and shown an exact picture of contents of the screen they should accept. The users then moved on to a series of login forms where they were either able to login, or press a button that stated the form was insecure. The auxiliary security screen would change at each page and the user's responses where tracked. About 2/3rds of the way through the test, users were told that instead of different types of screens, they would only see two screens: the proper screen and a screen that said the input was insecure. At the conclusion of the test, the users provided feedback on a paper survey.

In the first part of the study, the auxiliary screen displayed one of the following readouts during the study:

- A screen that read `firefox`. Users were instructed that this screen meant their input was secure.
- A screen that read `Firefox`. With the first letter capitalized. Users were instructed to press the button that said the form was insecure if the auxiliary security screen deviated from the lower case "firefox" readout.
- A screen that read `f1refox`. With the first 'i' replaced by a 1.
- A screen that read `internet explorer`. The name of a complete different, but related program.
- A blank screen.

In the second half of the test, users were given a screen that read out `-INPUT INSECURE-`. This interface of affirming an insecure state was compared to previous results where the insecure state was discovered by the lack of a proper security notification on the auxiliary screen.

During the experiment, the subject sat at a computer terminal with a researcher seated behind them jotting observations on a post-it. The researcher had a view of the screen, the auxiliary screen and the user's input. Users did not appear to

pick up on any signals from the observer as several thought they were doing badly when they were doing well or well when they were doing badly. The observer helped facilitate this neutrality by wearing a somber expression and a white lab coat.

3.4.2 Data

The first 10 questions were screened out of the data as a training period. Since the user responded slowly at first and slowly grew better through the first 10 responses, these responses were screened out as not representative. In addition, we also screened out outliers where users took greater than 30 seconds. This happened only on a few datapoints. The observation notes confirmed the majority of these instances occurred when the subject paused during the test to ask a question or receive clarification.

3.4.2.1 Cognitive Overhead Metrics

Cognitive overhead was based on a measure of time between responses. We present both the average responses in graph form as well as a few statistically significant statements that can be made about users cognitive overheads in one part of the experiment vs. another.

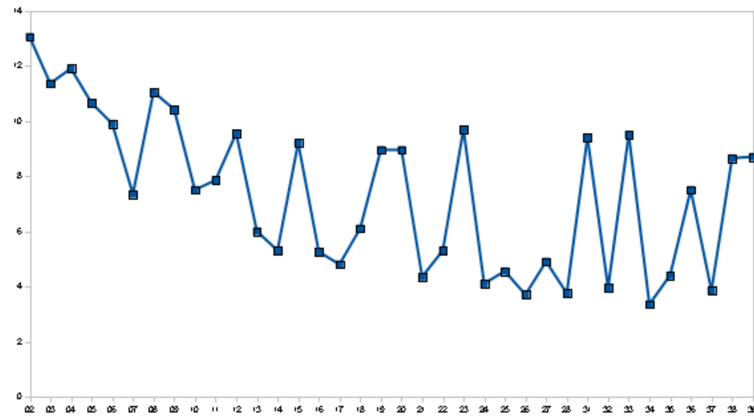


Figure 3.4: Graph of the averages

The following differences were all statistically significant:

- Users were quicker to respond during the phase of the test when the insecure state was affirmatively presented. (p=.029)

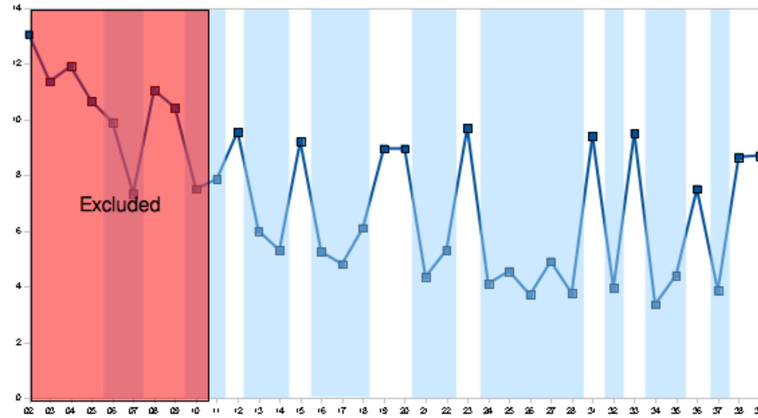


Figure 3.5: Graph of the averages, showing negative user responses shaded in blue and the excluded section of the results

- Users were quicker to determine that the system was insecure when presented with an explicit warning instead of a blank screen. (p=.017)
- Users were quicker to respond after the training period. (p¡.01)

3.4.2.2 Accuracy Rates

We present both raw and adjusted accuracy rates. The adjusted rates exclude data from three participants who produced particularly erratic and noisy data. These three participants either didn't correctly understand the experiment or mistakenly assumed that one of the fake screens was correct and the right screen was fake through all or a portion of the experiment. (People seemed inclined to prefer a capital F in their Firefox and would actually begin reject the lowercase one even though they were prompted to use that instead.) These types of mistakes would be unlikely to occur in a design like *LockBox*'s, so we excluded these particular cases in the adjusted results.

3.4.3 Findings

While security interfaces are tricky and often involve high rates of user error, our findings yield two different findings security interfaces should take into account to increase their success rate.

First, avoid interfaces which allow attackers to pick any strings on the security

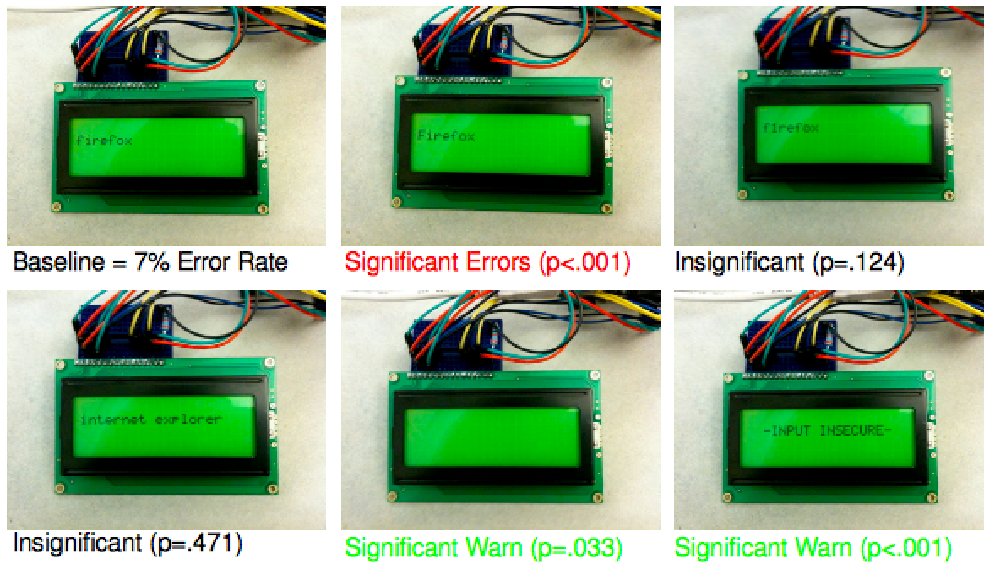


Figure 3.6: The accuracy rate compared to baseline for each security screen using raw data

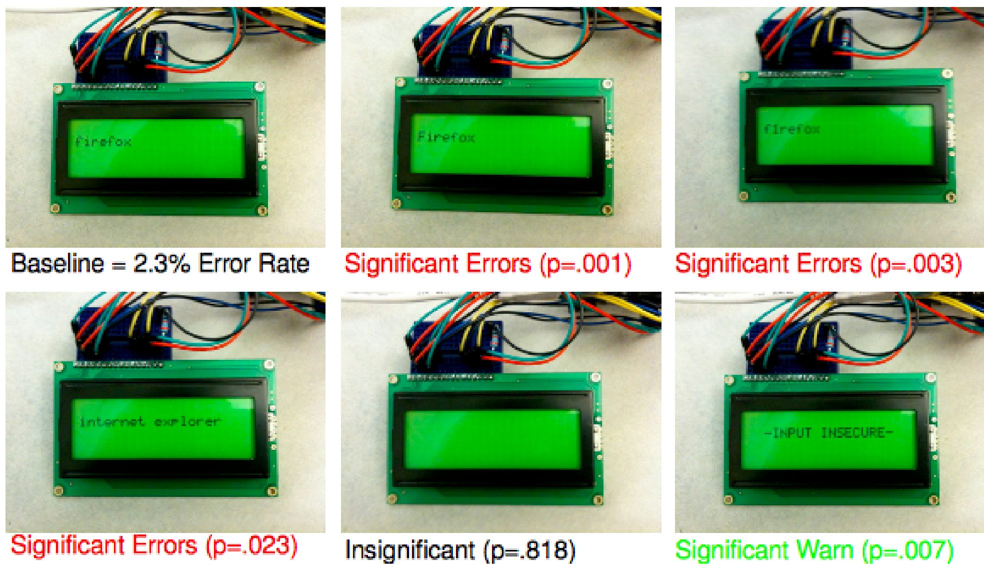


Figure 3.7: The accuracy rate compared to baseline for each security screen using adjusted data

indicator. Both the accuracy and cognitive overhead data that users were faster and more accurate when they did not have to distinguish between various similar looking

screens to determine which one was correct. This data supports the design approach made in *LockBox* to present users only with strings that they chose if a trusted channel is secure, or none at all. This design decision sets the interface in *LockBox* apart from interfaces which allow attackers to present users with similar name, such as in browser address bars in web browsers. The problems and challenges that present themselves for that type of interface are much greater.

Second, when the system is in an insecure mode, affirmatively display this information to the user. A blank screen was less effective than an explicit message. Users were both faster and more accurate if the screen displayed an explicit warning that their input was insecure. This is not as obvious a conclusion as it might seem. Having the security screen always display one message or another runs a risk of user fatigue in the long term. A longer term study needs to be done to determine whether affirmative displays of insecure modes continue to result in increased speed and accuracy over the long term. However, the short term data in this study is consistent with the idea that explicit notification helps users reach faster determinations.

These findings provide some hope that while security interfaces have often fallen short of providing the tools users need to effectively make good security decisions, better interfaces are possible. Accuracy rates increase significantly when interfaces avoid specific problems. Many existing systems have not been designed to avoid these mistakes, so continued poor results for these systems should not come as a surprise. With further study, security interfaces may one day achieve significantly higher levels of accuracy at reduced cognitive overhead.

3.5 When Encryption is Not Enough

Systems which provide protected contexts within an untrusted system, such as HARES [71], Overshadow [16], AEGIS [69], XOMOS [44] and others use memory encryption to protect the contents of memory. The goal of this protection is memory opacity, or the property that code which can only see the ciphertext of memory cannot determine anything about the contents of that memory. Unfortunately, memory encryption does not fully provide memory opacity. It can be shown any system relying on this assumption loses some degree of memory opacity over time. This class of analysis

can be called *temporal cryptanalysis*.

The starting assumptions include that an attacker can read only ciphertext, all encryption technologies are correctly and competently implemented and provide sufficient integrity checks to guard against writes. These checks are assumed to be durable over time, in that it is assumed that an attacker cannot use a previous block of ciphertext to revert memory to a prior value. These attacks are all problems, but fundamentally, there are cryptographic solutions to them.

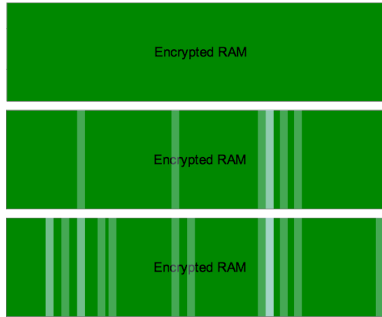


Figure 3.8: The ciphertext contents of RAM change as updates become visible

Temporal Cryptanalysis on the other hand, is driven by the understanding that data in systems isn't staic. While one observation of ciphertext yields little information about the contents of memory, additional observations rapidly begin to leak information about how the trusted code is changing, updating and interacting with memory. This information is useful.

None of this is particularly new. *Traffic analysis* is a very similar technique which often applies on networks and has been effectively used to break or weaken systems for decades. Traffic analysis is not unknown when it comes to protecting RAM either. It has been implemented using FPGAs on live memory busses to break real systems, notably in the reverse engineering of the Nintendo DS. This exact type of analysis has been noted in a previous work which built a system to try and mitigate these effects to some degree [26]. But somehow when it comes to trusted computing technologies, these risks are rarely discussed when the security properties of the system depend on memory opacity.

3.5.1 Proof of Concept

There doesn't seem to be a realization among the broader community that memory encryption cannot provide complete protection to memory contents. So maybe we need a better proof of concept. The goal of a proof of concept is a clear and easily demonstrated example of the flaw.

If we demonstrate a concrete example of an application's essential functions being seen and analyzed through the *changes* in memory data alone, we can show this problem is real in a more tangible way. Then we can simply rely on the hard won understanding the crypto community has demonstrated and embraced for decades: weaknesses only become more serious with time. Here we briefly present a proof of concept in a game of chess.

3.5.2 Tools for Temporal Cryptanalysis

One of the challenging aspects of developing a new proof of concept involves developing the tooling required for examining the problem. Often, new tools are needed to make things better. While plenty of existing memory snapshot tools do exist, we built our own targeting this particular issue:

- `memsnap` [23] is a memory snapshotting tool that provides the ability to dump the entire memory space of a process at a programmable interval. This allows us to compare observations at particular granularities. You can find it here: `http://github.com/djcapelis/memsnap`
- `memdiff` [21] is a memory differencing tool that outputs only the *differences* between subsequent memory snapshots with a programmable block size. You can find it here: `http://github.com/djcapelis/memdiff`
- `memxamine` [22] is a triage tool which provides a basic way of narrowing down the sections of memory that are most likely to be interesting to examine. You can find it here: `http://github.com/djcapelis/memxamine`

These tools work together to provide a toolset for exploring memory for these types of issues. If someone can show that `memsnap` and friends can produce a successful

cryptanalysis, then specifically written tools will be able to do even better with lower overhead.

3.5.3 Would You Like To Play A Game?

So how can we show that playing chess is vulnerable to this issue on systems which encrypt memory? In this case, we analyzed memory snapshots from xboard, [77] a gnuchess [32] frontend, running on a Linux system. Once we recorded snapshots of memory with `memsnap`, the problem became an issue of figuring out which sections were important to examine. Thankfully, there are some easy rules to triage our memory regions:

- Regions of memory that *never* change are uninteresting.
- Regions of memory that change *sometimes* are the most interesting.
- Regions of memory that *always* change are less useful.

In the version of xboard we examined, after looking closely at the regions which changed periodically and seemed to correlate with the times players made moves in the memory snapshots, multiple regions seemed ripe for exploitation:

- Offset 0x016e1a0 in memory region 0.
- Offset 0x01702a0 in memory region 0.
- Offset 0x016d6f0 in memory region 0.

These sections tended to change each time a move was made in our traces, but another section proved even more interesting. At offset 0x007fe20, a datastructure over

64kb long lies in memory. This structure updates exactly once per move and each move is recorded array style linearly in this memory section. Not only can one determine how many moves were made by seeing when this structure updates, if the memory writeback granularity is low, a clever attacker can recover the number of moves since the last observation of the ciphertext merely by looking at how much ciphertext changed. This means even if an attacker fails to make an observation after every move, they can still determine how many moves have been played.

And of course, given the rules of chess, an attacker that knows how many moves were made in a chess game also knows which player couldn't have won the game. (Technically they can't say the other player won, because the game might have concluded in a draw.)

You can see a youtube video of this proof of concept here: `https://www.youtube.com/watch?v=Eqrtn7LKuoE`

While this section of memory was particularly easy to exploit, it's important to note that our analysis highlighted *multiple* other memory ranges vulnerable to this type of analysis. This is not a problem with one particular datastructure in xboard. This is a problem with how transparent memory write patterns are to analyze while looking at changes in ciphertext.

3.5.4 Game Over: Attacker Wins

It is important for system designers to realize that memory encryption is a weaker form of protection than denying access to memory outright. While this may seem like a simple and naive proof of concept, it is especially important to note that this didn't require sophisticated machine learning algorithms, complex analysis or even memory snapshotting tools that are selective in what they capture. If it took sophisticated tools to show information can leak in memory encryption, that might be more comforting.

But it does not. With a general purpose memory snapshotting tool and a shell script, some of the most sophisticated trusted computing systems in the world can't protect the confidentiality of a chess game. Memory encryption is not sufficient.

3.6 Implementation Concerns

While *LockBox* was never completed as we designed it. If someone were interested in continuing this work, here are some concerns they might find relevant.

3.6.1 Hardware Cost

Several hardware alterations are required to the processor to implement *LockBox*. Due to the wide variation in other components of the system, such as the motherboard, memory interface and peripheral devices, I limit my analysis of *LockBox*'s hardware cost to the on-chip modifications.

The following on-chip modifications are present in LockBox:[4]

Caches The caches must be modified to store an additional flag of metadata. This flag indicates whether the line's security properties must be looked up in the Security Lookaside Buffer.

Security Lookaside Buffer What the TLB does for address translation, this new structure does for security constraints. This structure queries the memory arbiter to determine the security properties of the page.

Register File Due to the extremely sensitive timing of this component, it is critical the changes to the register file be as noninvasive as possible. The only alteration to the register file is that during a context switch, registers may need to be flushed by hardware. Instead of modifying the register file directly, a hardware assisted register flush can be implemented by running code from a small ROM. In this case, the register file itself will be able to remain unmodified with only the addition of a small ROM and minor control logic. Engineering challenges for each specific VLSI design will dictate the level of modifications possible for the register file.

3.6.2 Code Analysis

One way to begin to analyze the performance of systems like these without a complete prototype to run on is to look at the amount of code that one might split

[4] In this case I consider the memory controller, despite being on-chip in recent Intel processor designs, to be an off-chip component.

out as “security code” in various larger applications. In this section, I present data from a brief examination of a few applications to see what percentage of the memory accesses are attributed to encryption-related code. This analysis allows us to understand approximately how much additional overhead a system like *LockBox* might impose if the encryption functions of these applications were split off and run in a trusted context with some associated performance penalty.

Three applications were profiled:

- A text-based web browser (Lynx 2.8.6rel.4)
- An e-mail client (Evolution 2.10.2)
- An instant messaging client (Gaim 1.5.0)

Each of these applications was run under Valgrind [60] with the callgrind plugin. This tool can decompose all the application’s memory accesses (both code and data) and map them back to the function which caused them. These accesses were grouped together by library and the accesses caused by security libraries were separated from the accesses caused by the remainder of the program. For Lynx, the accesses from Libcrypto, a library providing SSL were considered “security code”. For Evolution, the accesses from LibSSL were considered “security code”. For Gaim, the accesses from the OTR library, providing off-the-record encryption for conversations, were designated as “security code”.

Program	Total	Secure	Percent
Lynx	74,159,190	10,297,322	13.885%
Evolution	5,341,809,490	669,341	0.013%
Gaim	12,660,199,842	569,526	0.004%

This provides a bound on the amount of accesses which would be exposed to additional overhead if one incorporated only the encryption functionality of an application into a trusted context. Fundamentally, the data supports the conclusion that security code is responsible for only a small portion of memory accesses in an application. Which means that if applications find the performance impact of trusted contexts unmanageable, splitting applications so that only the security-critical sections run inside a trusted context allow a very substantial section of code to run outside a trusted context without any associated overhead. For most applications, this seems unlikely to be necessary at

all, but for high performance applications, this approach may be a better alternative to eschewing use of security features altogether.

3.6.3 Deployment

Deployment is always a focal point of any change to core technologies. *LockBox*'s design has the following deployability advantages:

- *LockBox* gives control of security to the consumer buying the machine.
- *LockBox* can be deployed in software if compatible hardware is not available, or without additional layers of software if compatible hardware is used.
- Security threats against hypervisors and operating systems continue to increase. *LockBox*'s design provides users with a way to keep some security even when these layers cannot be trusted.

3.6.3.1 Developing the Software Ecosystem

After exploring the tradeoffs with *LockBox* in a hypervisor, and proving the feasibility of the hardware design, there remains a need to show the various types of security features that a system like this would enable. In particular, one of the most exciting areas of work is designing the modifications which will be needed to implement support for *LockBox* in most modern programming languages. There is lots of potential for integrating this work into mainstream compilers and making the security benefits of the system usable by adding a compiler flag during the compilation process. The compiler could emit a program that would invoke the *LockBox* framework and store secret data within the hardware. Since there is no requirement for cryptographic signing of programs, no certificates will be needed and nothing will need to be configured. Designing this type of high level interface would be key for achieving adoption of this technology.

There are several other interesting applications that could be developed with a prototype of this technology. Unfortunately the effort needed to develop these applications is beyond the time and resources of the initial rounds of my dissertation work. Future research projects could experiment with the following applications of the *LockBox* design:

Secure Input Libraries A modified input stack for a typical X11 session which can automatically request secure keyboard input for any trusted application which comes into focus. This could automatically provide secure keyboard input to any trusted application which asks for it.

Secure Linking and Loading The linking and loading environment may need to be modified to support some aspects of *LockBox*. New methods for using shared code will need to be found and some changes in this area will be examined.

Secure Web Browsers This application is the one which most inspired this design. The ability to type passwords into a web browser on public terminals without concern that the machine has been remotely programmed to store all keystrokes is something which should be available to all computer users.

Secure Virtual Machine Monitors With the increasing interest in virtualization and containerization technology, it would be interesting to take a virtualization or container framework and modify it to allow all of its guest systems to use this framework. This would allow the construction of a container system which, if compromised by an attacker, would not reveal data from the guest systems and be able to provide increased isolation for virtual machines.

Chapter 4

Final Thoughts

I chose a research area whose true endpoint was always going to be beyond the reach of my own efforts. These systems are big and change slowly. Computing systems touch a huge fraction of the global population. The cost of change is growing larger. It is amazing many of the systems we unquestionably rely on continue to work. Shared convention, understanding and mere tradition all play large roles in keeping these systems alive as they grow and evolve.

The explorations I've done to poke and prod at these systems and examine which parts might be best to change are my best guesses about how to improve our infrastructure. The ideas this document puts forth offer various tweaks and improvements. I suspect some of them will exist one day, in one form or another. Some of them exist already and I've tried to point out where they might interact with other ideas to create change larger than they would in isolation.

It's unclear how technology will unfold. It's unclear what influence, if any, the work I've done will have as these systems evolve. But what I've done here is collect, to the best I was able, a set of systemic evolutions worth implementing in some of the fundamental computing systems that run our world.

4.1 What Allows Change?

I'd like to take a moment to talk about the factors that drive change in technology. What leads people to adopt new technology? How does security tie in with

those motivations? What security concerns do people have around technology? What types of systems are most likely to bring about security improvements? It is in these questions that we can better understand how these proposals for change might succeed.

4.1.1 The Problem of Security

When evaluating choices, risk plays a role. Each person has a particular tolerance for risk. This tolerance determines when a choice becomes too risky for someone. Security describes practices to reduce risk. These practices can allow people to make choices they woudln't otherwise feel comfortable making, because the risk of those choices has been reduced or mitigated. Security changes are motivated when we realize that our tolerance for risk has been exceeded, or when we wish to be able to make choices that otherwise would exceed our risk threshold.

Unfortunately when it comes to technology, we've seen that people can often have a difficult time making security decisions. There are two main issues with our ability to make reasonable decisions about the level of security in our technology:

1. It is extremely difficult to correctly determine the risks involved in using technology.
2. When we design technology, we rarely state what level of risk is acceptable.

This does not match well with how humans deal with risk. Risk is something we not only tolerate, but welcome. While risk thresholds vary widely, people don't just adjust their level of risk downward. Risk compensation [68] [27] describes the human tendency to adjust our levels of risk upward as well. Since we adjust our level of risk frequently, it is very important to be able to judge our current level of risk.

Unfortunately with technology, this is rarely true. When we design our computing technologies, we don't articulate clear and reasonable risk thresholds. Our model of risk is binary, where something is either vulnerable or is not. At the micro level, this is often the reality we face. Yet, binary risk is hard for humans to engage with. We make choices to bring risk within our tolerance, but with binary risk, every act exceeds our tolerance. There is no middle ground.

This is in stark contrast with other fields, especially those with mature engineering practices. For instance, when we build a bridge, there are engineering documents

that state the estimated risk that the bridge may collapse and kill every individual on it. Mature engineering practices revolve around accepting this reality and calculating it. Obviously, we design bridges with the goal that they not collapse, but our ability to build safe bridges does not come from claims that the bridges we build will never collapse under any circumstance.

It should be noted that parts of our field live up to these standards. Encryption is a notable bright spot in security. We have many models to estimate the likelihood an encryption algorithm would fail. System reliability in VLSI design is another area where we are able to model risk accurately. Device engineers can tell you the likelihood that your calculation will be wrong given a certain intensity of cosmic ray strikes on a given circuit. Likewise, in storage, it is routine to model the likelihood of data loss or corruption on either an individual component in the storage stack, or across an entire storage system. Improving these models is part of the everyday work of research groups.

Yet other than encryption, most of security remains an area where we lack the ability to calculate risk effectively. Fundamentally, this is because so much in security involves human actions. Security exploits aren't things that happen to software by chance. Security exploits are things humans design and outcomes we achieve with a degree of intention. It is easier to predict whether or not an earthquake will cause a bridge collapse because we don't need to model an earthquake with malicious intent.

And the problem with all this is that when we can't calculate the risk, our ability to engage in risk compensation is weakened. Our ability to make informed decisions about the security design of our systems is flawed.

The biggest problem with security is that we don't know how much we need.

4.1.2 Beyond The Threat Model

If we don't know how much security we need, when do we make decisions to add it? Many of the changes in this dissertation were motivated by a desire to allow us to improve the security of our systems. How do we prioritize the addition of security against the other goals we have in advancing technology?

After all, the world is more complicated than decisions about security and risk. While security is an increasingly large problem, it is rarely the main motivation for

change in technology. Technology is a tool for humanity to increase our ability to cause change. Technology adoption tends to happen when it provides us new opportunities to interact with our world.

In this context, changing our technologies with the sole goal of achieving increased security is a difficult proposition. When a large technological change happens, we often call it a revolution. In a culture where widespread technological change shares the same terminology we use to describe government overthrow, we are forced to chose: do we want a revolution to improve security or to increase our ability to change our environment?

It is no wonder we rarely make large technology changes for the sake of security.

When we do make technological change for security, it is often when we feel under threat. While we are not very good at estimating our risks around technology, or how much security we might need in our systems, we do periodically notice when we have left the bounds of acceptable risk. This is one thing that inspires change.

We also increase security during times of change. Technology's track record with security systems that are combined with things that offer increased utility in other ways is much better. Various new systems from Facetime [6] to QUIC [70] are designing security features in from the beginning and increase security as those technologies are deployed for their other features.

So perhaps fundamentally the question is: can we design security technologies that also make technology simpler, easier and more powerful?

Bibliography

[1] GNU Screen Project. [Online]. Available: http://www.gnu.org/software/screen/

[2] Hacker News — OpenSSL is written by monkeys. [Online]. Available: http://news.ycombinator.com/item?id=1679926

[3] Linux PAM. [Online]. Available: http://www.kernel.org/pub/linux/libs/pam/

[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, p. 10.

[5] R. Anderson, "Cryptography and competition policy: Issues with trusted computing," *Workshop on Economics and Information Security*, pp. 1–11, 2003.

[6] Apple Inc. We've built privacy into the things you use every day. [Online]. Available: https://www.apple.com/privacy/privacy-built-in/

[7] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1997, p. 65.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 164–177.

[9] M. Belshe and R. Peon, "draft-mbelshe-httpbis-spdy-00: SPDY protocol," 2012.

[10] M. Belshe, M. Thomson, and R. Peon, “RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2),” 2015.

[11] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, “vtpm: virtualizing the trusted platform module,” in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[12] D. Bernstein. (2011) CurveCP: Usable security for the Internet. [Online]. Available: http://curvecp.org/

[13] J. Chandrashekar, Z. Zhang, Z. Duan, and Y. Hou, “Service oriented internet,” *Service-Oriented Computing-ICSOC 2003*, pp. 543–558.

[14] D. Chapman, E. Zwicky, and D. Russell, *Building internet firewalls*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1995.

[15] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, “Sharing and protection in a single-address-space operating system,” *ACM Transactions on Computer Systems*, vol. 12, no. 4, pp. 271–307, 1994.

[16] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports, “Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems,” *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2008.

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, “Planetlab: an overlay testbed for broad-coverage services,” *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, p. 12, 2003.

[18] C. Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen, “Protecting Systems from Stack Smashing Attacks with StackGuard,” *Linux Expo*, 1999.

[19] M. Cox, R. Engelschall, S. Henson, B. Laurie, E. Young, and T. Hudson, “Openssl,” 2001.

[20] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 221–232.

[21] D J Capelis. Memdiff. [Online]. Available: https://github.com/djcapelis/memdiff

[22] ——. Memdiff. [Online]. Available: https://github.com/djcapelis/memxamine

[23] ——. Memsnap. [Online]. Available: https://github.com/djcapelis/memsnap

[24] T. Dierks and E. Rescorla, "RFC 5246: The transport layer security (TLS) protocol version 1.2," Tech. Rep., 2008.

[25] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security.* New York, NY, USA: ACM, 2008, pp. 51–62.

[26] G. Duc and R. Keryell, "Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual.* IEEE, 2006, pp. 483–492.

[27] W. N. Evans and J. D. Graham, "Risk reduction or risk compensation? the case of mandatory safety-belt use laws," *Journal of Risk and Uncertainty*, vol. 4, no. 1, pp. 61–73, 1991.

[28] Facebook Inc. Facebook Login. [Online]. Available: https://developers.facebook.com/products/login/

[29] A. Freier, P. Karlton, and P. Kocher, "Secure Socket Layer 3.0," *IETF draft, November*, 1996.

[30] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 193–206, 2003.

[31] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis, “RFC 2764: A Framework for IP Based Virtual Private Networks,” 2000.

[32] GNU Chess Team. GNU Chess. [Online]. Available: https://www.gnu.org/software/chess/

[33] Google Inc. Google Sign-In for Websites. [Online]. Available: https://developers.google.com/identity/sign-in/web/

[34] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[35] IBM, “Software: Purify,” *IBM Rational*. [Online]. Available: http://www.ibm.com/software/awdtools/purify/

[36] Intel Corporation. (2010) Intel Advanced Encryption Standard Instructions (AES-NI). [Online]. Available: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/

[37] ——. (2012) Intel Ethernet Controller I350 Datasheet. [Online]. Available: http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-controller-i350-datasheet.html

[38] Internet Corporation for Assigned Names and Numbers. (2014) Remaining IPv4 Addresses to be Redistributed to Regional Internet Registries. [Online]. Available: https://www.icann.org/news/announcement-2-2014-05-20-en

[39] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux virtual machine monitor,” in *Linux Symposium*, 2007.

[40] E. J. Koldinger, J. S. Chase, and S. J. Eggers, “Architecture support for single address space operating systems,” in *ASPLOS-V: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 175–186.

[41] R. Kwan, R. Arnott, R. Paterson, R. Trivisonno, and M. Kubota, “On mobility load balancing for LTE systems,” in *Vehicular Technology Conference, 1988, IEEE 38th*, 2010, pp. 1–5.

[42] B. Laurie and P. Laurie, *Apache: The definitive guide.* "O'Reilly Media, Inc.", 2003.

[43] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *The Ninth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 35, no. 11, pp. 168–177, 2000.

[44] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[45] M. Lotter. (1988, November) RFC 1078: TCP Port Service Multiplexer (TCPMUX). [Online]. Available: http://www.rfc-editor.org/rfc/rfc1078.txt

[46] B. McCarty, *SELinux.* O'Reilly, 2004.

[47] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[48] D. K. Mulligan and A. Perzanowski, "The magnificence of the disaster: Reconstructing the Sony BMG rootkit incident," *Berkeley Technology Law Journal*, vol. 22, p. 1157, 2007.

[49] M. Peereboom. OpenSSL is written by monkeys. [Online]. Available: http://www.peereboom.us/assl/assl/html/openssl.html

[50] L. Peterson, T. Anderson, D. Blumenthal, D. Casey, D. Clark, D. Estrin, J. Evans, D. Raychaudhuri, M. Reiter, J. Rexford *et al.*, "GENI design principles," *IEEE Computer*, vol. 39, no. 9, pp. 102–105, 2006.

[51] W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange, "MinimaLT: minimal-latency networking through better security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM, 2013, pp. 425–438.

[52] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security and Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[53] D. Recordon and B. Fitzpatrick, "OpenID Authentication 1.1," *Finalized OpenID Specification, May*, 2006.

[54] J. Rutkowska and A. Tereshkin, "Bluepilling the Xen Hypervisor," *Black Hat USA*, 2008.

[55] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[56] J. Saltzer, D. Reed, and D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, p. 288, 1984.

[57] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3, pp. 157–170, 1972.

[58] E. Schultz, R. Proctor, M. Lien, and G. Salvendy, "Usability and security an appraisal of usability issues in information security methods," *Computers & Security*, vol. 20, no. 7, pp. 620–634, 2001.

[59] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[60] J. Seward *et al.*, "Valgrind, an open-source memory debugger for x86-GNU/Linux." [Online]. Available: http://www.ukuug.org/events/linux2002/papers/html/valgrind/

[61] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," *Proceedings of the 14th ACM conference on Computer and Communications Security*, pp. 552–561, 2007.

[62] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," in *16th Annual Conference on Computer and Communications Security*, 2009.

[63] S. Shepler, “RFC 2624: NFS Version 4 Design Considerations,” 1999.

[64] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai *et al.*, “BitVisor: a thin hypervisor for enforcing i/o device security,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.* ACM, 2009, pp. 121–130.

[65] B. Spengler, “Detection, prevention, and containment: A study of grsecurity,” *Libres Software Meeting*, 2002.

[66] R. Srinivasan, “RFC 1833: Binding protocols for onc RPC version 2,” 1995.

[67] P. Staubach, B. Pawlowski, and B. Callaghan, “RFC 1813: NFS Version 3 Protocol Specification,” 1995.

[68] F. M. Streff and E. S. Geller, “An experimental test of risk compensation: Between-subject versus within-subject analyses,” *Accident Analysis & Prevention*, vol. 20, no. 4, pp. 277–287, 1988.

[69] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, “AEGIS: architecture for tamper-evident and tamper-resistant processing,” in *Proceedings of the 17th annual international conference on Supercomputing.* ACM New York, NY, USA, 2003, pp. 160–171.

[70] The Chromium Project. QUIC, a multiplexed stream transport over UDP. [Online]. Available: https://www.chromium.org/quic

[71] J. Torrey, “HARES: Hardened Anti-Reverse Engineering System,” in *SyScan*, 2015.

[72] Trusted Computing Group, “Trusted platform module specification v1.2 rev103,” *Trusted Computing Group*, 2006.

[73] ——, “Trusted software stack specification v1.2 rev103,” *Trusted Computing Group*, 2006.

[74] Twitter Inc. Sign in with Twitter. [Online]. Available: https://dev.twitter.com/web/sign-in

[75] E. Witchel, J. Cates, and K. Asanović, “Mondrian memory protection,” *SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 304–316, 2002.

[76] W. Wong, “Stunnel: SSLing Internet Services Easily,” *SANS Institute, November*, 2001.

[77] xboard team. xboard. [Online]. Available: https://www.gnu.org/software/xboard/

[78] T. Ylonen and C. Lonvick, “RFC 4254: The Secure Shell (SSH) Connection Protocol,” 2006.