Rethinking Operating System Interfaces to Support Robust Network Applications

BY

W. MICHAEL PETULLO
B.S., Drake University, 1999
M.S., DePaul University, 2005

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:

Jon Solworth, Chair and Advisor
Daniel Bernstein
Jakob Eriksson
Tanja Lange, TU Eindhoven
Robert Sloan

To my wife Mary, who has—one way or another—accompanied me on all of the adventures of my adult life. This begins another.

ACKNOWLEDGMENTS

Working towards this degree has been a long journey and one that I could not have completed without the aid of many people along the way. I am indebted to those who made up for any of my shortcomings, including my advisor Prof. Jon Solworth; my dissertation committee—Prof. Robert Sloan, Prof. Daniel Bernstein, Prof. Tanja Lange, and Prof. Jakob Eriksson; and my fellow researchers in the Ethos laboratory, especially Pat Gavlin, Wenyuan Fei, and Xu Zhang.

Prof. Jon Solworth provided me patient mentorship and showed me what it meant to be a researcher. He spent countless hours discussing with me how to construct more robust systems, and he contributed tirelessly as I pursued my writing and presentations, including this thesis. Jon and I attended several conferences together, and he was always keen to introduce me to other researchers in the security and systems communities.

Ethos is Jon's project, though he claims that the project found him, instead of vice versa. Jon has been working on Ethos since 2007, and thus much of my work was simply to help refine ideas and concepts he had been developing for many years. When I arrived, it was clear that Jon had a design that looked at the problem of software robustness in a new way. Other students had completed much of the groundwork, including Ameet Kotian, who wrote the initial Ethos networking stack. Intrigued, I signed on, and I have found the resulting journey tremendously rewarding. I am glad to have contributed my small part.

I am also thankful to have been given the opportunity to work with Prof. Daniel Bernstein and Prof. Tanja Lange. Both are cryptographers, and both keep themselves incredibly busy.

Despite their tight schedule, Dan and Tanja collaborated with us on our Ethos network protocol work and volunteered to participate in my Ph.D. committee. I benefited tremendously from their participation, and I always had good conversations to look forward to when they passed through Chicago.

I was also inspired by Prof. Jakob Eriksson's *Introduction to Networking*—I hope to emulate the enthusiastic manner in which he designed and presented this course. Jakob also presented several lectures at our weekly *Advanced Programming Seminar*, and encouraged his students to do the same. In fact, one of Jakob's students, Tim Merrifield, set the attendance record for a lecture with an audience of 52. This record was later broken by ... Jakob.

Pat Gavlin laid much of the groundwork for what became Ethos' type system; his work as an undergraduate was without equal during the time we worked together. I was also lucky to work with Wenyuan Fei. As we continued the work on Ethos types, Wenyuan maintained the components necessary to construct Ethos applications in Go, and I focused on the code that would go into the Ethos kernel. I think we both found our efforts complementary, and I am grateful for the many discussions we had as we tried to integrate our components together.

Xu Zhang is another collaborator I am grateful to have worked with. Xu was instrumental in completing the port of Ethos to the AMD64 architecture, and this work greatly improved the performance of Ethos' networking stack. Since then, Xu has become our research group's architectural expert, fixing Ethos' memory protections on AMD64 and simplifying Ethos' interrupt handling.

ACKNOWLEDGMENTS (Continued)

Other students in our lab also provided aid. Yaohua Li ported Ethos' networking protocol to Linux. Siming Chen improved the performance of Ethos' memory allocator, further speeding up Ethos' networking. Divya Muppa, Francesco Costa, Vineet Menon, Fernando Visca, Giovanni Nebbiante, and Luca Cioria have all done work in Ethos user space, and provided valuable feedback that helped us improve Ethos' interfaces. Each student has also helped review papers and together they provided a critical audience for presentation rehearsals.

Santhi Nannapaneni serves as our department's Assistant Director for Student Affairs, and she runs her office with smooth efficiency. Her management of the department's qualifier program and the dissertation defense process serves students well and does much to sooth any anxiety. Sherice Steward was instrumental to the success of our *Advanced Programming Seminar*, as she made weekly orders for the pizza that accompanied our lectures.

Research in computer science revolves around our conferences. I was able to attend seven conferences/workshops with the help of generous travel grants from the Armed Forces Communications and Electronics Association, the United States Military Academy, the USENIX Association, the Association for Computing Machinery, and the National Science Foundation.

The US Army provided me a fellowship and time to pursue this degree. I would be remiss if I did not mention my fellow soldiers, many of whom deployed on one or more combat tours during my time here. I remain in awe of several of the men I previously was privileged to work with in Iraq and Afghanistan, and I am grateful for the education I received from them, whether superior, peer, or subordinate.

ACKNOWLEDGMENTS (Continued)

I am especially thankful for my parents, who throughout my upbringing made sure I understood the importance of education. My father set an example by his work ethic, and my mother provided a childhood that allowed my five siblings and me to dream. Both have influenced me immeasurably. As an undergraduate, John Zelle piqued my interest in systems and also provided patient direction, so I am glad I encountered him during this journey too. Above all, I am grateful for my wife Mary, who thankfully believes in me even when I myself cannot quite imagine where I am going.

Rangers lead the way!

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
API	Application Programming Interface
ARP	Address Resolution Protocol
CLR	Common Language Runtime
DAC	Discretionary Authorization Control
DAG	Directed Acyclic Graph
DH	Diffie-Hellman key exchange
DTLS	Datagram Transport Layer Security
DoS	Denial of Service
DoD	US Department of Defense
EAL	Evaluation Assurance Level
ECC	Elliptic Curve Cryptographic
ETE	Etypes Encoding
ETN	Etypes Notation
HAIPE	High Assurance Internet Protocol Encryptor
HVM	Hardware Virtual Machine
IDL	Interface Description Language

LIST OF ABBREVIATIONS (Continued)

IMF	Internet Message Format
IOMMU	I/O Memory Management Unit
IPC	Inter-Process Communication
IPsec	Internet Protocol Security
ISN	Initial Sequence Number
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LAS	Local Authentication Service
LEAP	Language for Expressing Authorization Properties
LoC	Lines of Code
MAC	Mandatory Authorization Control
MDA	Mail Delivery Agent
MIME	Multipurpose Internet Mail Extensions
MitM	Man-in-the-Middle
MinimaLT	Minimal Latency Tunneling
MMU	Memory Management Unit
NAT	Network Address Translation
NIST	US National Institute of Standards and Technology

LIST OF ABBREVIATIONS (Continued)

nonce	number-used-once
NSA	US National Security Agency
NSS	Network Security Services
ONC	Open Network Computing
OS	Operating System
OSKit	Flux OS Toolkit
PAM	Pluggable Authentication Modules
PFS	Perfect Forward Secrecy
PKI	Public Key Infrastructure
POTS	Plain Old Telephone Service
RAS	Remote Authentication Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SASL	Simple Authentication and Security Layer
SCTP	Stream Control Transport Protocol
SELinux	Security-Enhanced Linux
SHA-1	Secure Hash Algorithm
SSH	Secure Shell

LIST OF ABBREVIATIONS (Continued)

SST	Structured Stream Transport
TCB	Trusted Computing Base
TGS	Ticket Granting Service
TFO	TCP Fast Open
TID	Tunnel ID
TLS	Transport Layer Security
TPM	Trusted Platform Module
TTP	Trusted Third Party
UDP	User Datagram Protocol
UID	User ID
UUCP	Unix-to-Unix Copy
VM	Virtual Machine
VMM	Virtual Machine Monitor
SIP	Software Isolated Process
SMTP	Simple Mail Transfer Protocol
UUID	Universally Unique ID
XDR	External Data Representation
XSS	Cross-site Scripting

SUMMARY

This dissertation describes the network programming environment provided by Ethos, an operating system designed for security. Often, the interfaces provided by existing systems are very low-level. Experience shows that programmers on these systems have difficulty managing the resulting complexity when writing network applications. They must implement or integrate their own key isolation, encryption, authentication protocols, and authorization policies. Administrators must configure the same, often independently for each application.

Ethos eases the burden on application programmers and system administrators by providing more abstract interfaces and reducing code duplication. Instead of relying on applications to protect secret keys, Ethos keeps them in kernel space and allows their indirect use by applications through cryptographic system calls (e.g., sign). Ethos encrypts all network traffic and performs network authentication at the system level. Moving these protections to the operating system kernel allows Ethos to provide more informed access control, reducing the need for application-internal controls.

Thus Ethos provides a number of security properties unavailable in other systems. In many cases, Ethos application developers can write robust applications with zero lines of application-specific security code. Likewise, administrators do not need to learn application-specific configuration options. Instead, the majority of their work uses system-wide mechanisms, affecting all applications individually and the system in aggregate.

SUMMARY (Continued)

Many of the protections provided by Ethos sound straightforward to implement. However, we shall show that the system design that makes them possible is highly interconnected and not entirely self-evident. For example, how can Ethos authenticate at the system level when it is impossible for a system administrator to know every user that may be encountered on the Internet? In other cases, our design decisions became feasible only recently due to developments in hardware. Our hope is that our design appears clean, concise, and possibly—in retrospect—somewhat obvious.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Computer networking is ubiquitous. Its rise has enabled radical advancements in both communication and computing over the past few decades. Yet modern computer systems are facing a crisis. Networking has a dark side—it often exposes a computer to a population of attackers that spans the globe. This presents a problem both in that the sheer volume of attacks is staggering [1] and crimes easily span multiple, international jurisdictions [2]. The attackers range from novices to experts and often profit from selling their services [3; 4; 5]. Many appear to work for sovereign states, so their activities can be extraordinarily well funded. Thus attacks are often both effective and difficult to prosecute, and they result in damage ranging from personal inconvenience to matters of national security [6; 7; 8]. Experience indicates that current networked systems are unable to withstand the attacks that arise on the Internet.

How did we arrive at the current state of affairs? Let us first take a high-level view of system design, and then examine a critical property common to current systems. Computer systems are organized into strata—a layering that begins at hardware and progresses to an

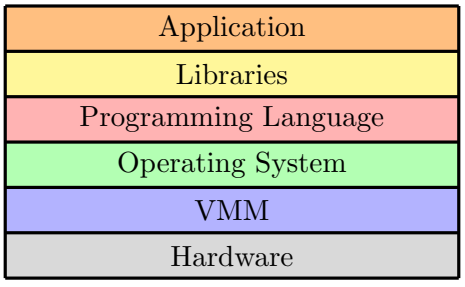


Figure 1: An abstract system strata

application. Each layer exports a set of interfaces to the layer above it. As we depict in Figure 1, these layers include hardware, an optional Virtual Machine Monitor (VMM), an Operating System (OS), programming language facilities, libraries, and applications.

A particular software strata is a reflection of many design decisions. For example, an architect may design an OS's system calls—which embody the interface exported to the stratum above the OS—with a very low level of abstraction. Essentially, this maximizes flexibility while imposing on higher layers (i.e., libraries and applications) the implementation of functionality not defined by the system calls. Saltzer et al. presented the end-to-end argument that favored this design in 1984 [9]. The authors reasoned that adequate performance and flexibility required that mechanisms such as bit recovery, encryption, and duplicate message suppression be deferred to applications: that they should be end-to-end. This consideration drove the design of today's mainstream OSs and the distributed systems built on them.

While an OS that provides a low level of abstraction (consistent with the end-to-end principle) maximizes flexibility, it increases the burden on application developers and administrators. This has many consequences. In POSIX, networking is very low level; it does not even provide for strong authentication. Such security functionality is left to applications or the libraries they build upon, and this leads to duplicating critical code. This duplication increases a system's attack surface. As an example, the Fedora Project's Linux distribution contains three major Transport Layer Security (TLS) C libraries: OpenSSL, GnuTLS, and Network Security Services (NSS). Fedora's effort to consolidate on NSS [10] and thus aid system assurance began in 2007 and is still ongoing, having faced many obstacles. Furthermore, libraries must often

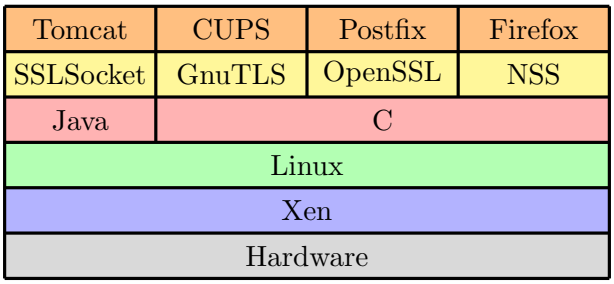


Figure 2: A particular system strata: end-to-end security

be rewritten for different programming languages. Thus a typical strata begins to resemble Figure 2. We call this the *end-to-end-security strata*, and POSIX serves as a commonly-used example.

Even if it were feasible to standardize on a single TLS library, the amount and complexity of code required to properly invoke the library is often non-trivial. Well-meaning and otherwise competent developers routinely misuse TLS libraries [11; 12], resulting in vulnerabilities. Georgiev et al. showed that an application from a major bank did not verify the identity certificates associated with its encrypted connections and thus was vulnerable to Man-in-the-Middle (MitM) attacks. In other cases, applications do have protections, but they can be bypassed by exploiting bugs in the application.

For a particular example of the subtle issues that arise from the end-to-end-security strata, we studied the Postfix email server. Postfix was written by a security expert, and establishes a rough limit on our ability to develop a robust application on POSIX. Yet even Postfix contained a flaw in its use of OpenSSL that resulted in the possibility of a plaintext injection attack [13]. Like Georgiev et al., we found that the volume of security-critical code necessary to invoke security libraries in POSIX applications is large, leaving much room for such errors. Postfix

contains around 2,000 lines of code to support robust networking (notwithstanding the code in the libraries themselves). Many other applications share this pattern: The Dovecot IMAP server, Apache, and the third-party mod_auth_kerb Apache module contain 15,000, 1,800, and 1,500 network-security-related lines of code, respectively. Apache's mod_ssl is made up of 11,700 Lines of Code (LoC). Moreover, fixing an application like Postfix does not preclude other applications from having the same flaw. In fact, Pure-FTPd and the Cyrus IMAP server were later found to have the same plaintext injection vulnerability as Postfix [14; 15].

The HiStar authors state: "*experience has shown that only a handful of programmers have the right mindset to write secure code, and few applications have the luxury of being written by such programmers*" [16]. We believe that an end-to-end-security strata provides application programmers and administrators too many opportunities to make network security errors. On an end-to-end-security strata, security protections must be present in each application. Here such protections can be:

(1) missing due to application programmer omission,

(2) missing due to application misconfiguration, or

(3) negated by exploiting an application bug.

We call the class of violations that can result from these vulnerabilities *application-based subversion.*

Experience shows that it is simply too difficult to develop large, robust network applications when significant security responsibilities fall on each application. Instead, we advocate an

increased reliance on the natural OS property of *complete mediation*, whereby the OS verifies the authority of *every* object access [17]. The *OS kernel* is the portion of a computer's software that runs in the processor's privileged mode, that is, during execution it has access to the processor's privileged instructions. In contrast, *programs* or *applications* are that portion of a computer's software without direct access to privileged instructions. Thus programs rely on the OS kernel to mediate access to hardware and kernel data structures. This is most often done through the use of a *system call*, a type of procedure call that invokes a kernel entry point.

The OS is the first software layer that completely mediates all program accesses to external resources [17]. This includes communication between processes, with hardware devices, and with other hosts on a network. Because of this, the OS is in a unique position with respect to security. Properly designed and implemented, its controls are absolute; they cannot be bypassed by applications. For this reason, we propose that an OS should provide basic, universal, and threat-appropriate protections, and application-specific end-to-end protections should only add to this baseline.

Luckily, technological advances redefine the requirements which lead to end-to-end design. In many cases, performance is no longer significantly affected by the presence of strong protections. In fact, multi-core processors often sit idle. Network latency—not the encryption that once taxed processors to provide secret communication—now dominates performance. Many researchers have thus demonstrated that universally encrypting communications is feasible [18; 19]. Even many resource-constrained, mobile devices can process encryption at Gb/s rates [20]. Moreover, the rise of commodity virtualization also affects the end-to-end principle.

The "end" can now be an OS instead of an application. Multiple OSs running on a single computer give rise to tailored OS interfaces that can maintain appropriately rich but consistent semantics for all of their applications.

1.2 Introduction to Ethos

Ethos is an OS designed to provide a level of robustness commensurate with modern threats. To minimize complexity, Ethos forgoes backward compatibility [21] and provides system calls with semantics that benefit the security of the entire system. This resembles how application security is affected by programming-language semantics. For example, the problem of fixing application buffer overflows is solved for all applications by using a strongly-typed programming language. This strategy is much more effective than using a programming language without type safety and trying to avoid buffer overflows in each application. Similarly, OS system call semantics can have a profound impact on application robustness. Of course, this idea has been applied to OSs before: a similar philosophy drove the move from cooperative to preemptive multitasking. (There still are particular systems which benefit from deterministic cooperative multitasking; likewise, we expect Ethos to be suboptimal for some non-security-critical applications.) Yet while programming language semantics continue to evolve (e.g., Haskell, Erlang, Go, Scala), OS semantics have been relatively unchanged for decades [22]. Ethos reduces opportunities for application-based subversion by providing a maximal set of protections at the OS stratum while remaining general enough for modern secure applications.

Ethos targets Virtual Machines (VMs) instead of bare metal. A VM-based approach means that other OSs can coexist with Ethos. This allows Ethos to avoid the *application trap*, in

which no one will use a new OS because it has no applications and there are no applications because there are no users. Instead, it is reasonable to use Ethos even for a single application. More importantly, Ethos can remain focused on security, while other OSs can provide semantics appropriate for other application requirements (i.e., VMs refine end-to-end as previously mentioned). We describe some of the benefits and drawbacks of our VM-based approach in more detail in Appendix D.

Ethos currently provides memory paging, processes, protected networking, types, and a filesystem. We have completed 39 system calls and several related Go packages. On this, we have built a shell, basic tools, a remote shell utility, and a networked messaging system. We have ported Go and Python to Ethos for the purpose of writing applications. We have also ported C, but only to implement the OS and support other programming languages.

1.3 Threat model

Chapters 4–9 will describe Ethos' security properties; here we describe the threats we designed Ethos to resist. We are broadly concerned with the confidentiality, integrity, and availability of network communication. The attacker we consider obviously will try to directly intercept communications to violate our protections, perhaps by defeating protocols [23; 24], assuming control of network media, deploying counterfeit services, or making odd requests to Ethos-hosted services. He might also attack a system more indirectly, by running applications on a target Ethos host (including malicious software), controlling remote hosts, or subverting other unprivileged virtual machines. He should not be able to use these capabilities to violate network security either. For example, an application buffer overflow on Unix can cause

the application to misuse TLS, but we will describe why Ethos' strata makes such a result impossible.

Denial of Service (DoS) attacks from *known* users are expected to be addressed by a host's authorization policy or non-technical means. An *anonymous* attacker might try to affect availability, through transmission-, computation-, and memory-based DoS. An attacker with enough resources (or control over the network) can always affect availability, so we attempt to drive up his costs by making his attack at least as expensive as the cost to defend against it. Here we want to address equally non-MitM and MitM attackers. That is, the ability to spoof the source IP address of a packet *and* capture a reply should not allow much easier attacks.

Although an attacker might have a local user account, we explicitly trust system administrators; thus they are outside of the threat model. We do not entirely ignore an attacker's ability to have physical access to a computer's components. While such attacks are exceedingly difficult to defend against, we attempt to mitigate them. Ethos' consolidation of cryptographic operations in the kernel makes it easier to enforce the use of side-channel-resistant cryptographic implementations, track and destroy keys, and enforce the use of Trusted Platform Modules (TPMs) [25]. Most importantly, defenders must also consider physical security and personnel issues.

If an attacker does gain complete control over a client or server, through physical access or otherwise, he will be able to decrypt very recent and future packets, but, given some symmetric key lifetime i, he should not be able to decrypt packets older than i nor identify the parties involved in this communication. We refer to this as approaching *Perfect Forward Secrecy*

(PFS). True PFS implies $i = 0$, but, for performance reasons, i is generally positive but sufficiently small. PFS is important because it limits the damage resulting from a computer that is physically compromised.

1.4 Outline of the work

The central hypothesis of the Ethos project is that careful software design can guarantee that certain security properties hold even if we cannot predict future applications. Here two critical questions are:

- *What design and implementation properties must a system have to thwart the adversaries described above?* And, of course,
- *What particular design will guarantee these properties?*

The first question is addressed by the strength and assurance literature. The second is the main motivation for this work. Thus our focus complements the substantial and far better understood work on *implementing* reliable OSs, for example by microkernels [26; 27; 28; 29; 30; 31].

Trusted Network Interpretation [32] defines several network-related security services, which we refer to as requirements R1–R7:

R1: Authentication identifies the subjects that interact with a system.

R2: Header and data integrity (communication field integrity) means that a system protects data present in network communication from unauthorized modification. This includes header fields and application data.

R3: Non-repudiation provides unforgeable proof of submission or receipt of data. In particular, *non-repudiation of origin* and *non-repudiation of receipt* can be separate properties [33].

R4: Denial of Service resistance (continuity of operations/protocol-based DoS protection mechanisms/network management) means a system will continue to provide service to legitimate users even when an adversary attempts to consume a disproportionate number of system resources. In particular, both protocols and network management facilities must be designed to be robust against DoS attacks.

R5: Data confidentiality protects against unauthorized disclosure. Here we are most concerned with the ability of an adversary to passively observe data that one computer transmits to another over a network.

R6: Traffic flow confidentiality is concerned with disrupting traffic analysis attacks, which derive information from message length, frequency, or participants.

R7: Selective routing means that messages may be differently routed as requirements change. The necessity for this might arise because network links are unavailable, links pass through the control of an adversary, or the cost of a link changes.

A general-purpose robust networked system must address each of these requirements. In addition, hosts must implement the protections required by the *Trusted Computer System Evaluation Criteria* [34]; of particular interest to networking is authorization (**R8**). For each of these security responsibilities, we describe why existing solutions fall short in Chapter 2.

We have mentioned that the OS kernel mediates all access a program has to its environment, including users, hardware, other programs, and remote services. Ethos' foremost strategy is to use complete mediation to make network-security-related protections universal. Ethos brings several security responsibilities into the kernel, making them compulsory from the point of view of programs and thus removing opportunities for application-based subversion. The universal properties provided by Ethos are:

P1: Network authentication Ethos assigns every user (and process) an immutable, unforgeable Universally Unique ID (UUID), even in the cases of previously unknown or anonymous network users. No user can assume another user's UUID without first compromising a secret cryptographic key, and neither does a compromise of a network authentication server compromise these keys.

P2: Network encryption Ethos encrypts *all* application-layer network messages and a maximal amount of lower-network-layer data.

P3: Type checking Ethos subjects all application input and output to a type checker so that no application receives ill-formed input or produces ill-formed output.

P4: Key isolation Ethos isolates secret cryptographic keys so that they are never shared with applications. Furthermore, several of Ethos' features help ensure user keys are used only with user approval.

P5: Denial of Service resistance Ethos' network protocol provides built-in DoS protections.

TABLE I: Comparison of network-security requirements with Ethos security properties, along with a comparison with POSIX

Assurance literature requirement	**Ethos protection**	**Comparable POSIX implementation**
R1	P1	Left to application code
R2	P2, P3	Either IPsec or left to application code
R3	P4	Applications have access to secret cryptographic keys
R4	P5	Protocol-dependent, generally weak universal protections
R5	P2	Either IPsec or left to application code
R6	P2*	e.g., IPsec tunneling or (for strong protections) Tor
R7	n/a†	n/a†
R8	P6	Integration between system-wide authorization systems and application-provided network authentication is difficult

*Additional protections delegated to applications due to performance requirements
†Left to lower-level protocol processing (i.e., networking layer)

P6: Network authorization Ethos' authorization system does not allow unauthorized users to interact with user-space application code in any way.

Traffic flow confidentiality is partially addressed by Ethos by protecting host-to-host connections within a single cryptographic tunnel, but additional protections (e.g., white noise or onion routing such as provided by Tor [35]) are not made universal because of the overhead they impose. We consider selective routing a characteristic of the network layer, which exists below Ethos' networking. Table I compares the assurance literature requirements to the protections provided by Ethos.

After presenting an overview of Ethos' networking Application Programming Interface (API) in Chapter 3, we describe Ethos networking from the bottom up, starting with Ethos' network

protocol. Ethos' rich system calls place certain demands on its network protocol, including key negotiation, encryption, and remote-user authentication. It is not possible for an Ethos application to transmit cleartext data over the network, so Ethos must perform well—maximizing throughput and minimizing latency—despite its strong protections. We found that existing network protocols do not meet these requirements, so we designed Minimal Latency Tunneling (MinimaLT). MinimaLT is a transport-layer network protocol whose structure provides integrity and confidentiality guarantees while minimizing latency. Chapter 4 describes MinimaLT.

Every security evaluation class described in *Trusted Computer System Evaluation Criteria* [34] addresses identification and authentication. This should come as no surprise—after all, *if you don't know who you are dealing with then how do you decide what to allow them to do?* In Ethos, all network requests are authenticated before any application code runs. Chapter 5 focuses on authentication in Ethos.

Types have long been enforced in programming languages to ensure higher quality programs. We have applied type safety to interactions between programs by enforcing type safety at key points in the Ethos kernel. Ethos files and Inter-Process Communication (IPC) channels have a type associated with them, and Ethos ensures that no program reads or writes ill-formed data with respect to these types. We describe this in Chapter 6.

Signed certificates provide more restrictive properties than general-purpose authentication and are used to authorize especially sensitive operations. For example, I may wish to request a withdrawal of $200 from my bank account in such a way that the bank could not repeat the

request without my approval at a later time. Likewise, my bank should demand reasonable assurance that precisely this request did, in fact, come from me. Chapter 7 will describe Ethos' unique approach to certificates.

Ethos provides Mandatory Authorization Controls (MACs). Like Security-Enhanced Linux (SELinux), Ethos considers both the user and executing program when making authorization decisions. Such fine-grained controls are critical to thwart our attacker, who may try to fool users into running malicious programs. As we will describe, Ethos has relatively few system calls, and this simplifies authorization policy specification. This differentiates Ethos authorization from SELinux, which is complex [36; 37] partly because Linux's system call interface is complex and low-level. In particular, Ethos can make authorization decisions based on a remote user because network authentication is performed by the OS, and Ethos can restrict the production of certificates and other critical types. Chapter 8 focuses on Ethos' authorization system.

Finally, we present a series of case studies in Chapter 9 and conclude in Chapter 10. As we conclude, we summarize the lessons we have learned, provide a summary of Ethos, propose directions for future research, and summarize the software resulting from this work.

CHAPTER 2

PREVIOUS WORK

We begin our discussion of previous work by describing strength and assurance, because this literature presents what is required of a robust system in a broad sense. Such work begins to answer the question: *What design and implementation properties must a system have to thwart the adversaries described in Chapter 1?* Following that, we present a survey of the network-specific security services required to satisfy these requirements (§2.2). Next we describe some particular authentication and protected network protocols (§2.3). Finally, we describe a series of existing OSs that integrate security services and protocols in an attempt to satisfy strength and assurance requirements (§2.4). These represent particular designs, but we show that their state-of-the-art security-mechanism compositions often fail to provide a robustness commensurate with the threat found on the Internet.

2.1 Robustness

The *strength* of a system design refers to how much effort an adversary must expend to defeat a perfect implementation of the design [38]. A high degree of strength is difficult to obtain; the US National Security Agency (NSA) spent ten years designing and analyzing the strength of VINSON, a specialized voice encryption device [39]. Adequate strength in more general systems is even harder to achieve. Clearly a tension remains between strength requirements and the pace of innovation many markets demand.

Much work has been done to examine the requirements of a high-strength design. Published in 1985, the US Department of Defense (DoD)'s *Trusted Computer System Evaluation Criteria* [34] (the Orange Book) provides security criteria and methods of evaluation that may be used to produce strong computer systems. Here we define four key terms found in the Orange Book—the security policy, assurance, a system's Trusted Computing Base (TCB), and covert channels—and then we summarize the requirements set forth by the Orange Book.

A *security policy* is the set of constraints an organization operates under in order to preserve the organization's continued effectiveness, and such policies must reflect the broad operational goals of the organization. Thus violations to the security policy—assuming the security policy is proper—cause harm to the organization. Such harm might threaten safety, privacy, financial posture, or infrastructure [38]. Security policies are organization-specific, but generally address confidentiality, integrity, and availability. Of course, a draconian security policy is not proper—such a policy itself reduces an organization's effectiveness. Non-technical policy includes hiring practices and physical security, and policies historically have been enforced by procedures, organizational culture, and rules of law. Here we are interested in the technical application of a security policy as a set of rules that govern a computer system.

Assurance provides confidence that a system is correctly implemented and that its security mechanisms enforce an organization's security policy. *Life-cycle assurance* applies to system design, development, and maintenance. Through each phase of a system's development, the system must be guarded against unauthorized changes, and it must be reevaluated when changes occur that could undermine its original assurance. *Operational assurance* minimizes the like-

lihood that a system's protections could be circumvented during use. Thus a highly assured system of strong design provides a high level of *robustness*, that is, it continues to function properly (i.e., does not violate the security policy) even in the presence of an intelligent adversary.

A system's *TCB* is the collection of components responsible for the enforcement of the security policy, despite the presence of other, untrusted components. That is, a good TCB minimizes the chance that an attacker can use (untrusted-) application-based subversion to violate the security policy. A TCB commonly includes hardware, firmware, and both OS and application software. (We avoid the alternative name *security kernel* [40; 41], because a TCB/security kernel is generally broader than the similarly named OS kernel, as defined in Chapter 1.) Lampson points out that it is not entirely true that a correct TCB denies untrusted components the ability of violating the security policy—untrusted components remain able to affect availability. Thus Lampson claims that it is enough for the TCB to be *fail-secure*: a trusted component that fails may cause the system to deny access that should have been granted, but never grant access that should have been denied [42]. A TCB should be as small as possible so that it can be carefully assured.

A *covert channel* is a residual communication medium that can be used to violate the security policy. The Orange Book classifies covert channels as either storage- or timing-based. In *storage covert channels*, one process stores information that is later directly or indirectly read by another. *Timing-based covert channels* use the timing of events as a signal. Covert channels can be very difficult to remove from a system completely because many are a result

of decisions necessary to achieve adequate performance; reasonable countermeasures include reducing the bandwidth of covert channels or detecting their use.

The Orange Book describes the broad requirements of four assurance divisions: D–A. D provides minimal protection, C provides discretionary protection, B provides mandatory protection, and A provides verified protection. Each division contains one or more classes, designated with a numeral (e.g., C1).

Each security level in the Orange Book addresses six fundamental security requirements:

(1) A system must identify the subjects with which it interacts (authentication).

(2) Objects must bear some type of access-control label.

(3) A system must allow for the specification of a policy that regulates the access of objects by subjects (authorization). Systems reference object labels and subject identities when making these authorization decisions.

(4) A system must maintain accountability information, so that an audit can identify the individuals who performed security-related actions.

(5) It must be possible to evaluate a system to establish the sufficiency of the previous requirements' implementation.

(6) It must be impossible to subvert these requirements.

What makes the Orange Book's strictest division, A1, unique is the high degree of formality it requires. A1 requires a formal model of a system's security policy and a mathematical proof that the model supports the policy. In addition, the system must carefully define its TCB, show

that its TCB supports the system's security policy, verify the implementation of the TCB, and formally analyze the system to identify covert channels.

Trusted Network Interpretation [32] (the Red Book) applies the Orange Book notions to the domain of networked systems and describes several services unique to networking. The classes of services described by the Red Book are communication integrity, denial of service, and compromise protection. These classes contain the protections introduced in Chapter 1: network authentication, header and data integrity, non-repudiation, denial of service resistance, data confidentiality, traffic-flow confidentiality, and selective routing. We discuss each of these in more detail in §2.2.

The International Organization for Standardization (ISO) *Common Criteria for Information Technology Security Evaluation* [43] is a framework for specifying security requirements and then implementing and evaluating them. The Common Criteria assigns Evaluation Assurance Levels (EALs) that indicate the level of assurance achieved. These levels range from EAL1, functionally tested, to EAL7, which requires a formally verified design.

Ultimately, high assurance levels—above the EAL4 rating of today's widely used OSs [44]—are possible only if a system has low complexity and security is part of its initial design. Unfortunately, existing systems are very complex and were designed in a time of low security requirements. They have accreted functionality over time, resulting in extraordinarily difficult-to-understand systems and, therefore, security holes.

2.2 General security services

We now present a series of general security services which satisfy the requirements set forth by the strength and assurance literature. This includes cryptographic primitives, which provide the basis of many security services; traffic-flow confidentiality; DoS resistance; authentication; authorization; and type safety.

2.2.1 Cryptographic primitives

Cryptography most often provides network header and data integrity, non-repudiation, and data confidentiality (R2, R3, and R5). We discuss three cryptographic primitives—encryption, hashing, and signatures—because each is used in Ethos. Encryption protects secrets and has a long history [45]. Typically, a small cryptographic *key* and arbitrary *plaintext* serve as input to an *encryption* algorithm or *cipher*. This algorithm produces *ciphertext* as output. Given some ciphertext, it should be infeasible to transform the ciphertext back into plaintext without access to a particular secret (either the key used to encrypt or a key mathematically-related to the encryption key). An attacker could try to enumerate all of the possible keys, but a typical 128-bit key might be any of 2^{128} possibilities. Thus ciphers are specialized enough that their strength can be stated in a straightforward way: a cipher provides 128-bit security if breaking the cipher is equivalent to finding the correct key from 2^{128} possibilities.

The use of a key by ciphers is a convenience because the same key may be used to encrypt many messages. The problem of protecting a large secret—the sequence of messages—is replaced with protecting a small one—the key. Only the key (and plaintext) must remain secret;

the cipher itself can be well-known. This is known as Kerckhoffs' principle [46], and it allows ciphers to be subjected to rigorous examination.

In reality, an encryption algorithm or its implementation might contain a flaw that reduces the number of keys that could have produced a given ciphertext. Such flaws reduce the work an attacker must do to find the corresponding plaintext. Often, such flaws are very subtle. For example, some Advanced Encryption Standard (AES) implementations succumb to *side channel attacks*, whereby an attacker reduces the number of possible keys by observing the effect of memory caches [47]. Other attacks include observing power usage during cryptographic operations [48]. Careful implementation—including introducing random, input-independent delays; ensuring operations' inputs do not affect their timing; or using data masking [49]—can mitigate these attacks, but such protections reduce performance. Other ciphers avoid this severe performance loss because they are resistant in their default, high-performance implementation to software-based side-channel attacks.

Cold boot attacks provide another threat to key isolation [50]. These attacks exploit the data retention properties of DRAM to extract information from a memory chip after turning off a computer and physically removing its DRAM. The attacker can extend the duration of the chip's data retention and thus increase the likelihood of success by decreasing the temperature of the chip. Zeroing memory is the most obvious countermeasure for cold boot attacks (e.g., Miller et al. mention limiting the existence of plain-text passwords [51]), but some keys are necessarily persistent and often no single program is aware of where in memory all keys reside.

Ethos uses ciphers that belong to one of two categories: symmetric or asymmetric. With *symmetric* ciphers, both parties participating in a communication have a copy of the same key. They use this single key to transform plaintext to ciphertext and vice versa. With *asymmetric* ciphers (*public-key cryptography*), each participant maintains a pair of mathematically related keys [52]. One is kept secret and the other is published. If one key transforms plaintext to ciphertext, the other reverses the operation. Asymmetric ciphers tend to operate more slowly than symmetric ciphers and also require a larger key size to achieve the same strength. Despite this, their properties are necessary for many cryptographic constructions. For example, Diffie-Hellman key exchange (DH) allows two parties to negotiate a shared secret after providing each other with their public keys. (This subsumes the need to distribute $O(n^2)$ secret keys using a separately secured channel to support pair-wise-isolated communication amongst n parties, assuming each party knows the identity associated with each public key.) We use a particular variation of DH, Elliptic Curve Cryptographic (ECC) DH [53], in Ethos. We discuss another application of asymmetric ciphers, cryptographic signatures, below.

A cryptographic *hash* function takes as input arbitrary text t and produces a fixed-length hash h. Such a function H is designed so that it is not feasible to (1) given h, determine the original input t to H that produced h; (2) given t and h, find some other text t' such that $H(t) = H(t') = h$; or (3) produce some u and u' such that $H(u) = H(u')$. Uses of cryptographic hashing include integrity checks, naming, and cryptographic signatures.

A cryptographic signature function takes as input a user u's secret asymmetric key k_u and an arbitrary message m and produces a *digital signature*. The digital signature serves as proof

that u endorsed m. This property of non-repudiation holds as long as u maintains the secrecy of k_u, the correspondence between u and k_u is known, and signatures are generated using k_u only when directed by u. Public-key ciphers such as RSA provide a convenient means to produce digital signatures [54]. First, u hashes m to produce h. Next, u encrypts h using his secret asymmetric key. Then u publishes this ciphertext along with m. Someone who wants to verify the signature transforms the ciphertext back to h using u's public asymmetric key, hashes m, and compares the results. If the comparison matches, then the signature is valid.

2.2.2 Traffic flow confidentiality

Even encrypted messages can leak information. During World War I, the German Army used a cipher called ADFGVX for communications between corps-and-above-level headquarters. Although the Allies had not broken ADFGVX, they we able to predict several German offensives merely due to an increase in the number of transmissions that used the cipher [55]. The Red Book states that message length, frequency, and protocol fields (such as IP addresses) disclose useful information to an adversary. Transmitting white noise can blind message lengths and frequency, but decreases the throughput of the network.

Network-layer protocol fields are difficult to hide, as they are necessary for the delivery of a message. Tor [35] provides robust network anonymity through the use of *onion routing*. In onion routing a host chooses a path of relays, $r_1, r_2, \ldots, r_n$, with corresponding keys $k_1, k_2, \ldots, k_n$, between it and a message's destination. Before transmitting, the host encrypts the message once using each of $k_n, k_{n-1}, \ldots, k_1$ before transmitting it to r_1. Each relay decrypts the message with its key, revealing the next hop, and then passes on the partially decrypted message. Eventually

the message is transmitted to its final destination, but (assuming $n \geq 2$) no r knows the entire message path or the message contents. Another technique is to simply broadcast a constant signal that alternates between encrypted messages and random data to all possible recipients. This is the presumed purpose of *number stations*, which transmit continuous streams of numbers using shortwave radio [56].

2.2.3 Denial of Service resistance

A *DoS attack* violates system availability. Here an attacker's requests overwhelm a host, leaving its remaining resources insufficient to service requests from other parties. There are three categories of network DoS attack: network exhaustion attacks, memory exhaustion attacks, and computational attacks. A single host cannot thwart an attacker with overwhelming resources [57], but countermeasures can protect against attackers with fewer resources. DoS countermeasures are worthwhile even against a successful attacker, because they consume an attacker's resources and thereby limit the number of victims.

The countermeasure for network exhaustion attacks requires increasing the throughput of the network, although a protocol can avoid contributing to such an attack by removing the opportunity for *amplification attacks*. In an amplification attack, the attacker makes network requests that elicit a larger response, exploiting a side effect of a protocol's design. Such an attacker could spoof the origination address of his requests, causing a large amount of traffic to be directed to a third party by the server, while requiring little traffic from the attacker.

Memory exhaustion countermeasures include making a protocol stateless until after authentication is complete. TCP's three-way handshake provides a weak form of authentication, but

early implementations required a host to maintain information even before the handshake was complete. This allowed an attacker to use a *SYN flood attack* to cause many half-open connections, eventually filling a server's connection state buffer and causing the server to become unresponsive. With *SYN cookies*, a server does not need to maintain information about half-open connections [58; 59]. A SYN cookie encodes a timestamp, minimum segment size, and the hash of the client IP address, client port number, server IP address, server port number, and a secret value into the Initial Sequence Number (ISN) included in a SYN/ACK reply. Because TCP requires that a client's ACK use the sequence number $ISN + 1$, the encoded information allows a server to validate an ACK without having stored any information from the original SYN.

Puzzles provide a countermeasure to computational DoS attacks [60; 61]. Here a server requires that a client solve a computational puzzle before the server will perform work for the client. This causes the client to expend computational work before it can cause a server to do the same. For example, in Chapter 4 we will describe a puzzle whereby a server calculates $h = \text{hash}(n)$ and provides a client h and n', where n' is n with its i rightmost bits replaced with zeros. The client must solve the puzzle by finding n. It does this by enumerating through the possible values for the i missing bits, calculating the hash of each candidate for n, and comparing the result to h. Thus the client must perform $O(2^{i-1})$ hash operations.

2.2.4 Authentication

Authentication identifies *subjects*—users, hosts, and other entities—with which a system interacts. Authentication is fundamental to security protections because it is a prerequisite for

authorization. Without knowing who a person is, a system cannot determine what he should be allowed to do. That is, without authentication, every subject must be treated equally. Systems, of course, must be able to treat subjects differently. After all, it is likely that an administrator should be allowed to do different things than a secretary. A CEO will have different access requirements than an accountant.

Each subject has one or more identities, referred to as a *principal*. Some users assume a recurring principal, but keep secret their real-world identity; we call these users *strangers*. Other users produce a new principal for each request; we call these users *anonymous*.

Authentication must be strong. It must limit the occurrences of *fraud*—where someone fools the system to masquerade as another user—and *insult*—where a legitimate user is rejected access [62]. Passwords are a weak form of authentication; their use for network authentication has been fraught with problems due to brute force attacks, spoofing, keylogging, and network timing attacks [63; 64; 1]. Cryptographic techniques are much better suited for use on a network, and local authentication can be strengthened through the use of physical tokens or biometrics.

Network authentication typically is mutual, whereby a client authenticates a server and the server authenticates the client. Without *mutual authentication*, a MitM attack is possible. In this attack, an adversary positions himself so that he intercepts messages between two parties before rebroadcasting each of them. Using strong, mutual authentication to negotiate an encrypted channel renders a MitM attack exceedingly improbable.

2.2.5 Authorization

Once a subject is authenticated, its access to system *objects* is governed by *authorization*—the application of the security policy. Objects include resources like files, network services, and devices. An object might have an *owner*, a closely associated subject. Authorization may be *discretionary*—defined by an object's owner—or *mandatory*—immutable except for by external controls.

A *reference monitor* [65] is a key component of classical authorization systems. A reference monitor implements an authorization predicate that takes as input a security policy and various information describing a subject, request, and object. Given this input, a reference monitor outputs either accept or reject. An OS typically will consult its reference monitor before servicing a system call. If the reference monitor rejects a request, then the OS will return an error to user space. Otherwise, the OS will continue to process the system call.

2.2.6 Types

We now turn to a general protection that has a broad effect on distributed system integrity. Type safety prevents *untrapped errors*—where an error goes unnoticed and computation continues—and can reduce *trapped errors*—where an error is detected and computation stops [66]. Untrapped errors are particularly pernicious since they result in arbitrary behavior; this arbitrary behavior occurs when a program's runtime state becomes inconsistent. Deep implementation knowledge and extensive analysis can be required to predict the execution after an untrapped error. For example, C's lack of type safety permits untrapped errors such as

buffer overflows [67; 68], and the end-to-end-security strata increases the likelihood that these errors will result in security holes.

Thus programming language designers have long incorporated type safety to reduce the effort of writing applications. Milner stated "*well-typed programs cannot 'go wrong'*" [69]. While surely an exaggeration, strongly checked type systems increase program quality.

Programming languages can be *untyped*, in which a variable can hold values of any type; or they can be *typed*, in which each variable is restricted by a type [66]. Type-safe programming languages have no untrapped errors. This is ensured either by *dynamic checking* in untyped languages (or certain language features) or by *static checking* a typed program prior to execution. Go is a typed language, LISP is an untyped language, and assembly is a language that is both untyped and unsafe.

Specialized types in programming languages are useful for preventing particular errors, and in general depend on programming language semantics and tightly-coupled code. Examples include linear types [70] which can ensure at most one reference to each object, and union types [71] which can allow performing only operations valid for two or more types.

Closely related to type safety is *object serialization* whereby a programming language object is encoded to an external form for storage or transmission. Serialization is necessary in a distributed system to ensure data passing between components has a consistent form. Mechanisms for serialization range from the basic (e.g., htons) to the sophisticated. Care must be taken when serializing an object containing pointers which might produce *shared objects*—where two or more objects each reference a given object—or *cyclic objects*—where an object contains

a direct or indirect reference to itself. A standard technique meets these requirements by maintaining a table containing indices and object references [72]. Each time an object is encoded, the table is checked to determine if the object had been previously encoded. If it had been, the corresponding index is encoded instead of the object. A similar process is used while decoding to recreate object references.

In an *implicitly typed* encoding, only data is encoded, not its type. This reduces encoding size and eliminates type field interpretation while decoding. In *explicit typing*, types are encoded along with data. Serialization schemes can be either closely coupled to a programming language or programming-language-agnostic.

Remote Procedure Calls (RPCs) are procedure calls that are executed in another address space [73]. RPCs build on serialization to provide a natural interface into remote services, and the abstraction they provide can make programs easier to understand. Some RPC systems allow communication with remote computers and others—often for performance reasons—allow communication only within a single computer [74; 75; 76].

Python [77], Java [78], C♯ [79], and C++'s Boost [80] provide examples of native serialization facilities. RPC systems in particular benefit from programming-language-specific serialization. The homogeneity of Java's Remote Method Invocation (RMI) [81] makes it easier to use than heterogeneous systems since it does not need a programming-language-agnostic Interface Description Language (IDL). Programming-language-specific systems can also provide conveniences such as the ability to pass previously unknown objects such as subtypes to remote

methods. This enables polymorphism (in other systems a subtype might be interpreted as its parent at the distant end).

External Data Representation (XDR) [82], ASN.1, JavaScript Object Notation (JSON) [83], and Protocol Buffers [84] can serialize objects from any programming language. Google designed the latter after experience demonstrated settling on one programming language was infeasible. Open Network Computing (ONC) RPC builds on XDR [85]. The rpcgen utility generates client stub and server skeleton code from a high-level declaration, and programmers fill in the rest. CORBA provides an any type and inheritance [86], and identifies actual types using a type code. CORBA's type codes are ambiguous [87], and this ambiguity can violate the type safety of inheritance and any types, particularly in loosely-coupled systems. Thrift [88] is designed to be particularly flexible. Instead of dictating a single encoding format or transport protocol, Thrift exports an interface with which to implement them.

2.3 Protected networking

Protected networking builds on the security services we have discussed to provide:

(1) a means to identify remote parties through authentication and

(2) a means to maintain confidentiality, integrity, and availability of the communication channels between parties.

Our discussion of network authentication covers Kerberos, a symmetric-key-based system that is commonly used by enterprises, and public-key certificate systems. Following this, we describe a number of network protocols.

2.3.1 Kerberos

Kerberos was motivated by the transition from single, time-sharing systems to distributed networks of workstations. The authors of Kerberos wanted to produce an authentication system that:

- allowed users and hosts to prove their identity;
- was resistant to eavesdropping and replay attacks; and
- was transparent (now referred to as *single sign on*), that is, a user presents a password once and from that point on can authenticate to any service on the network.

Kerberos is an authentication system built around symmetric key cryptography and the Needham-Schroeder cryptographic protocol [89]. (The Needham-Schroeder protocol was broken in 1995, but the protocol was fixed in a subsequent revision [23].) At a minimum, a Kerberos installation is made up of two services: an authentication service and a Ticket Granting Service (TGS). In addition, there is client software, most commonly in the form of a library that is linked into client applications. Each subject maintains a secret key that is shared only with the Kerberos authentication service.

Kerberos defines two constructs that are used for authentication: tickets and authenticators. Kerberos issues tickets to subjects who use them to prove their identity to end servers; tickets have a lifetime that allows them to be used repeatedly. (A ticket is valid for a single subject, server pair.) Subjects use authenticators to prove that they are the entity to whom Kerberos issued a ticket. Authenticators may be used only once, but subjects may self-generate them.

We use the following syntax to describe the details:

a	Network address of the client workstation
A_C	An authenticator generated by C
c	A client identifier
K_S	A symmetric key known only to Kerberos and S
$K_{C,S}$	A symmetric key known only to Kerberos, C and S
l	Ticket lifetime, chosen by the ticket issuer
M	a message
s	A server identifier
t	A timestamp
$T_{C,S}$	A ticket that identifies C to S
$\boxed{M}_{K_{C,S}}$	Encrypt message m using symmetric key $K_{C,S}$

$$T_{C,S} = \boxed{s, c, a, l, K_{C,S}}_{K_S}$$

Figure 3: A Kerberos ticket that client c may present to server s; note that it is encrypted with the server's key, thus it could not have been directly generated by c

Figure 3 and Figure 4 list the contents of a ticket and authenticator, respectively. Figure 5 depicts how Kerberos authenticates a user:

(1) The authentication process starts with a user entering his name at a client workstation. The client sends this name to the Kerberos authentication server.

$$A_C = \boxed{c, a, t}_{K_{C,S}}$$

Figure 4: A Kerberos authenticator, generated by client c for server s; the server knows this authenticator comes from client c because only server s, client c, and Kerberos have $K_{s,c}$

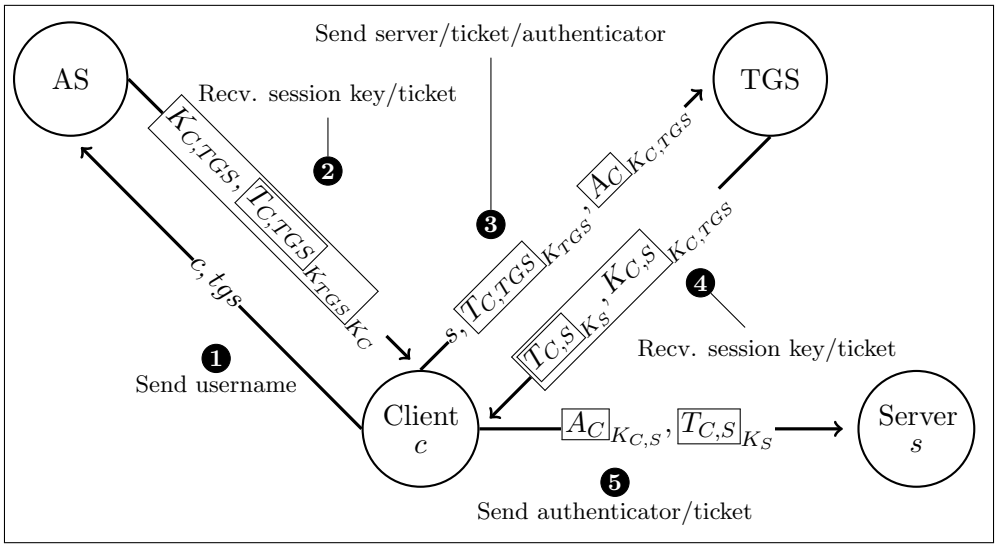


Figure 5: Kerberos authentication protocol (based on an existing figure [90])

(2) If the user is valid, the Kerberos authentication server responds with a tuple, encrypted with the user's cryptographic key. This tuple includes a session key and ticket for use with the Kerberos TGS. The ticket arrives encrypted with the TGS's key. (Thus the user cannot generate, view, or modify the ticket.)

(3) The client application now prompts the user for his password, hashes this password into a cryptographic key, and uses this key to attempt to decrypt the response from the authentication server. The client sends to the TGS the end server to which the user wishes to authenticate; the encrypted ticket from the authentication server; and an authenticator, encrypted with the session key from the authentication server.

(4) The TGS decrypts the ticket and confirms s, c, a, and l. If these fields are valid, the TGS extracts $K_{C,TGS}$ from the ticket and uses it to decrypt the authenticator. If this authenticator is also valid, then the TGS responds to the client with another ticket. This ticket is encrypted with the end server's key K_S, contains the session key $K_{C,S}$, and is for use with the end server.

(5) Finally, the client sends the ticket and an authenticator to the end server.

In this protocol, Kerberos must only bother the user for his password once. Successive connections to services can be negotiated without the user's input (i.e., single sign on). Eventually, the ticket granting ticket received at step two will expire, and only then will the user be required to enter his password again.

Kerberos is built upon symmetric instead of public-key key cryptography. This has two negative effects: First, a compromise of the Kerberos service itself is catastrophic. From that point on, an attacker can masquerade as any user because he has all of the subjects' secret keys. Second, Kerberos does not scale beyond the enterprise because all parties must trust a single Kerberos installation. This is not sufficient for the Internet, where such a universally trusted third party does not exist.

Another problem exists in many client implementations. In such applications, cryptographic algorithms and keys often share the same address space as application logic. This increases the risk that a key will be compromised, and it means that an improperly written, improperly configured, or compromised application might not perform authentication properly. That is, most Kerberos implementations give rise to application-based subversion of authentication.

2.3.2 Certificates

Certificates are digitally signed statements [42]. The properties of certificates make them useful for solving identity problems in distributed systems. A system that produces a certificate represents the *signer*, and the *relying party* verifies and possibly acts on a certificate. A primary goal of such a system is that the signature scheme is *unforgeable*, meaning only user U can create U's signature on message m, and *universally verifiable*, meaning any user can verify that the signature on m is valid [91]. Together, these provide the property of *non-repudiation of origin* [33]. Thus certificates provide much more precise authentication than systems such as Kerberos. With Kerberos, a system identifies a remote user, but can go on to claim the user made a request that he did not. In contrast, certificate-based systems can require an unforgeable record of every request.

We discussed digital signatures as a cryptographic primitive in §2.2.1. However, ensuring the integrity of a certificate system takes more than cryptography. To provide non-repudiation, a system must:

(1) provide a correspondence between users and their private cryptographic keys,

(2) ensure adequate isolation of private cryptographic keys,

(3) guarantee that the system signs only what the user intends, and

(4) guarantee that the signer and relying party share a single meaning for a given certificate.

An *identity certificate* is a signed statement that vouches for a subject's identity, addressing (1). For example, the certificates used by web browsers are signed statements that map public

keys to servers. To authenticate using certificates, a system first requests an identity certificate from the party making a request, and then determines if the direct signer of the identity certificate is trusted. If this is not the case, then the system obtains an identity certificate for the signer, and repeats this process until it encounters a signer it trusts (or it gives up and rejects the original request). We refer to this final signer as a *Trusted Third Party (TTP).* The path from the original signer to the TTP is a *certification path*. Once a system knows the identity associated with a key pair, it can accept or reject any type of certificate generated using that key pair based on its corresponding identity.

A difficult question is: *who makes an appropriate TTP?* It is notable that when two parties perform a mutual authentication, they need not rely on the same TTP. This is a second key advantage of certificate systems over systems like Kerberos, and it means they have the potential to scale to the entire Internet. In practice, web browsers use X.509 certificates and often designate universally trusted parties, but researchers have recently noted the insufficiency of this technique [92; 93]. A strong certificate system requires that relying parties choose their TTPs. X.509 added a graph-based trust model to its traditional hierarchical model [94], but its design imposes a high performance overhead. SDSI [95] also provides a strong trust model, but likewise does not perform well at Internet scale. Another alternative is the web of trust used by PGP[96].

Many systems have tried to isolate private keys in user space in an attempt to address (2). For example, Secure Shell (SSH) [97] attempts to isolate private keys by protecting them with an optional password and requiring restrictive file permission settings. In §2.4.3 we will

describe Plan 9 [98], which improves on the approach of SSH and makes it system-wide. But such user-space isolation is often compromised by malicious software [99] because malicious software runs with the user's privileges and passwords are optional or the password choice is at the user's discretion. It is difficult to close these potential security holes when individual applications implement disparate protection systems.

External smart cards are often used to provide stronger isolation. A typical smart card is a physical device that contains a cryptographic key pair k; a cryptographic processor that, given some input m, can sign m using k to produce a signed output s; and an interface that allows writing m and reading s (without giving access to k). Sometimes, smart cards also provide a trusted input or output channel; this can help address (3), although it may significantly increase production cost.

Beyond isolating keys, a system must ensure that they are used only as a user intends. A system does this by prompting the user or forcing him to enter a PIN to unlock his smart card. But current software layering is very complex, making this difficult. It is possible to fool common OSs and web browsers into either revealing a secret key or authenticating using client-side SSL without notifying the user [100]. Even if the user is warned that a signature is about to take place, users will often ignore security warnings [101].

When user approval controls can be bypassed, isolation attacks can exist even on systems that use smart cards. In *keylogging/PIN collection*, an attacker collects user input, possibly including a smart card PIN, after installing malicious hardware or software. This can result in fraudulent authentication, whereby a malicious program authenticates to a computing resource

using a user's credentials; *fraudulent signatures*, whereby a program performs a digital signature without a user's approval; and *remote smart card control*, whereby a program makes the smart card's services available to third parties over a network [102]. In *keyjacking*, an attacker undermines the notification mechanism of a certificate system, allowing certificate operations to occur without notifying the legitimate user [100]. This violates the requirement that a user's private key is only used with his approval.

Even assuming a trusted means of reviewing material, it may be difficult to know whether to sign it. The *semantic-level difference* is the difference in cognitive understanding of the meaning of a digital document between the signer and the relying party. The *syntax-level difference* refers to the representation of data at the syntax level, between two points within a system [103]. Many attacks may exploit a semantic difference. For example, a *replay attack* exploits a certificate whose meaning is insufficiently clear; a signed copy of a vague statement is dangerous because it lacks context [104]. A *Dalí attack* exploits a polymorphic file; a single file displays different contents depending on what application is used to view it [105]. For example, a single file may contain data which is valid both as PDF and TIFF. Selecting a program capable of displaying one format or the other will determine which of the possibly unrelated representations gets displayed. Furthermore, it is possible to digitally sign a document that references external material. If this is done without care, one can alter the external material without invalidating the digital signature [106].

2.3.3 Protected network protocols

Protected network protocols should provide header and data integrity, data confidentiality, and DoS resistance without burdening application developers. Furthermore, protected network protocols must perform well, that is, their protections must not impose an unacceptable performance penalty relative to unprotected protocols. Here we discuss several existing protocols and describe why they do not satisfy these requirements.

TLS (previously Secure Socket Layer) provides cryptographic network protections above the transport layer [107]. TLS is widely deployed as the primary network security layer in web browsers. A typical use of TLS uses ephemeral DH to produce a session key, RSA-based certificates to provide authentication, AES to provide channel encryption, and Secure Hash Algorithm (SHA-1) hashing. Collectively, these represent an example of a TLS *cipher suite*.

We depict establishing a new TLS connection using a client-side certificate on top of TCP/IP in Figure 6. A full connection begins with a three-way TCP handshake, followed by the TLS messages described below:

(1) The SSL connection begins with a ClientHello message. This allows the client and server to negotiate a protocol version number and cipher suite.

(2) In its ServerHello response, the server also provides a session ID, provides a certificate, and requests a client certificate.

(3) The client responds with its certificate, a cipher-specific value, a certificate authenticator, and an indication that it is switching to authenticated and encrypted transmission.

(4) The server responds, also indicating it is transitioning to protected communication.

Including the TCP/IP handshake, but not a DNS lookup, this process takes three round trips to complete.

Figure 7 depicts an abbreviated TLS connection, made possible by the session ID. A client may include a session ID in its ClientHello to indicate that the client wants to resume the parameters previously negotiated (1). If the server agrees (2), this reduces the number of round trips necessary to establish the TLS connection to two because application data can accompany the Finish message (3). This shortcut also reduces the number of expensive public key operations required when using RSA-based certificates.

TLS is most often implemented as a user space library, and this leaves its authentication, integrity, and confidentiality protections susceptible to application-based subversion. Many applications across the Internet still omit the use of TLS or use a weak cipher suite. One survey found that 77% of websites using HTML's password field are exclusively served by cleartext HTTP [108]. Furthermore, many TLS libraries have confusing APIs with insecure defaults, and even well-meaning developers routinely misuse them [11; 12]. To authenticate their peer, an application must: validate a certificate path, check the hostname present in the server's certificate, and check that the certificate has not been revoked. Often these checks are performed incorrectly. Developers have also been known to deactivate certificate checking during testing and then forget to turn it back on prior to release.

Beyond the difficulty of using TLS correctly in every application, TLS incurs a latency overhead of at least one round trip, in addition to the latency of TCP/IP. Datagram Transport

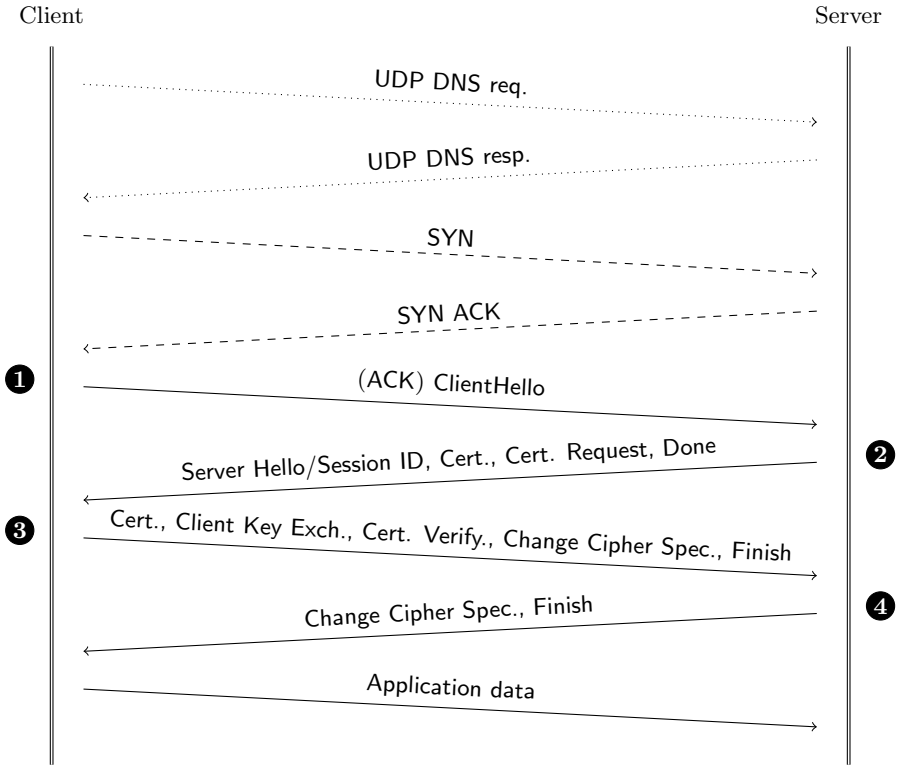


Figure 6: A full TLS connection establishment requires three round trips before transmitting application data; the dotted round trip is only necessary if the host's IP address is not yet known

Layer Security (DTLS) [109] provides TLS protections on top of UDP, but it shares the latency of TLS' cryptographic handshake (and, of course, is not stream-oriented). There have been many attempts to reduce this latency. Recently, False Start, Snap Start, and certificate prefetching have accelerated establishing a TLS session [110; 111; 112].

Latency improvements have also been attempted within the layers below TLS. TCP Fast Open (TFO) [113] allows clients to request a TFO cookie they can use to forgo the three-way handshake on future connections. To benefit from TFO, a server application must be idempotent: TFO does not detect duplicate SYN segments, so the application data included in

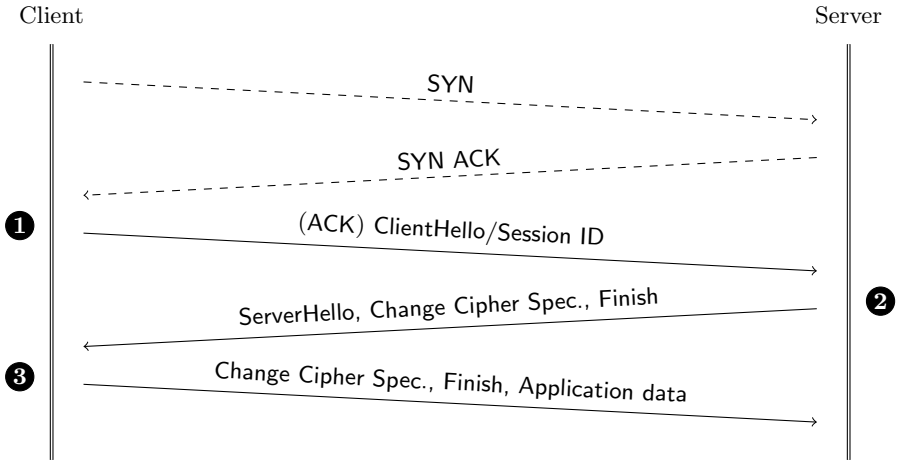


Figure 7: With an abbreviated TLS connection establishment, application data may be sent after two round trips

such segments must not cause the server to become inconsistent. Furthermore, since any client may request a TFO cookie, a client may spoof its sending IP address and mount a DoS attack against a server by requesting cookies; under this condition, the server must again require a three-way handshake.

Tcpcrypt [19] investigated ubiquitous encryption, but it maintains backwards compatibility with TCP/IP. Tcpcrypt provides hooks that applications may use to provide authentication services and determine whether a channel is encrypted. Our approach is different; we are interested in a clean-slate design that centralizes authentication and encryption services in an OS's system calls to ease assurance.

Internet Protocol Security (IPsec) provides very broad confidentiality and integrity protections because it is generally implemented in the OS kernel. For example, IPsec can be configured such that *all* communication between node A and node B is protected. This universality simplifies assurance; in fact the NSA employs a variant of IPsec in its High Assurance Internet

Protocol Encryptor (HAIPE) interoperability specification [114]. IPsec's major shortcoming is that its protections stop at the host; it focuses on network encryption and host authentication/authorization. For example, IPsec does not authenticate or restrict users across the network. It is certain that recent, high-profile attacks on US Government networks at some point passed through IPsec-protected networks [7; 6].

Stream Control Transport Protocol (SCTP) is a transport-layer protocol that provides reliable delivery and congestion control [115]. SCTP differs from TCP in that it can bundle messages from multiple applications (i.e., *chunks*) into a single packet. Structured Stream Transport (SST) [116] allows applications to associate lightweight network streams with an existing stream, reducing the number of three-way handshakes incurred by applications and providing semantics useful for applications that use both data and control connections.

2.4 Systems

Here we discuss existing systems that attempt to apply the security services and protocols we have previously described to satisfy strength and assurance requirements. As we discuss each, we will point out their deficiencies, eventually describing how Ethos improves on them in Chapters 4–9.

2.4.1 Multics

Multics was an early OS that investigated how to develop more secure operating systems [117]. Multics was designed as a time-sharing system and "*provides computation and information storage service to a community of users*" [40]. As such, it did not investigate to a large extent the security problems that arise from computer networks. Despite this, the contributions

of Multics to host security are large, in particular its contribution to hardware-based memory paging and segmentation; its hardware-based separation of kernel space from user space; its access controls; its extensive use of a compiled language, PL/1; and its five design principles [118]. The Multics design principles are:

(1) Grant permissions conservatively, that is, assume access is not permitted until it is granted.

(2) Subject every object access to authorization controls (i.e., complete mediation).

(3) In a broad sense, security should not require the system's design to remain secret (similar to Kerckhoffs' principle).

(4) Every program and user should be granted the least privilege necessary to complete their authorized task.

(5) Provide a natural, easy to use interface so that users are not motivated to bypass the system's controls.

Multics provides a hierarchical filesystem that is governed by access control lists. Processes serve as subjects and can access objects in the storage system. Each subject has associated with it a value called a principal identifier, which corresponds to the user on whose behalf the process runs. Each object in the storage system has associated with it three modes, read, write, and execute. For each mode, there exists a list of principal identifiers that may access the object using the mode.

Multics principal identifiers have structure, they are in fact *multicomponent principal identifiers* made up of three components:

(1) an individual user name,

(2) a group called a *project*, and

(3) a group called a *compartment*.

Project administrators determine project membership, but compartments are permissive: users may select any compartment to work within. Users choose which (permitted) project and compartment they will assume when they login to Multics. Multics specifies the user Bob, working in $project_x$ and $compartment_y$ as:

$$bob.project_x.compartment_y.$$

An access control could reference the principal identifier above, or it could use wildcards. For example, the following specification grants read and write access to any user that has been granted membership in $project_x$:

$$*.project_x.* \quad \text{rw}.$$

Multics access controls are discretionary. The owner of an object is free to set its access controls. Saltzer notes that this is insufficient for some applications; for example, Multics is unable to restrict a student so that he cannot set the permissions on his work so as to share it with other students [117]. This is the general problem with discretionary authorization: compliance with a security policy requires active cooperation from all system users.

2.4.2 Unix

Ken Thompson, a Multics veteran, went on to develop UNIX with Dennis Ritchie et al. [119]. Many of the lessons learned while developing Multics were applied to UNIX, although key differences did arise. Similarities include that UNIX was a time-sharing system; provided hardware memory and kernel protections; and was implemented primarily in a compiled language, C. A key difference is that UNIX filesystem objects are not regulated by access control lists. Instead, each object has a user and group associated with it, along with permissions for the object's user, the object's group, and everyone else. (Modern implementations have added access control lists to this basic mechanism, in general a trend that has increased the complexity of UNIX systems.) Two key contributions of UNIX were the pipe—including its integration into the UNIX shells—and the notion that many system objects were accessed using a single file-based API.

Like Multics, UNIX authorization traditionally has been discretionary. Throughout the 1990's several UNIX vendors added MAC to UNIX to produce compartmented mode workstations [120; 121]. Later, the NSA published SELinux, a mandatory access control system for Linux (a UNIX-like OS) [122] that focused on enforcing an authorization policy on a single machine. SELinux assigns each process a role, name, and domain. Objects—such as files, sockets, and pipes—have a name, role, and type. SELinux consults a configurable policy to govern how processes may access objects. SELinux is able to restrict the ports, IP addresses, and network interfaces a given process may interact with, but this level of control is insufficient for distributed systems.

```
1 listenFd ← socket (domain, type, protocol)
2 bind (listenFd, address)
3 listen (listenFd)
4 do forever
5     netFd ← accept (listenFd)
6     // Read and write socket netFd.
```

(a) A Berkeley sockets server

```
7 netFd ← socket (domain, type, protocol)
8 connect (sock, address)
9 // Read and write socket netFd.
```

(b) A Berkeley sockets client

Figure 8: A Berkeley sockets application in pseudo code

As a time-sharing system, UNIX did not originally emphasize networking as we know it today. UNIX-to-UNIX Copy (UUCP) provided networking support for early UNIX systems, but in a way more reminiscent of point-to-point Plain Old Telephone Service (POTS) serial links. Later, BSD added Berkeley sockets to UNIX [123], and this interface was eventually adopted by POSIX. The Berkeley sockets API includes five core system calls: socket, connect, bind, listen, and accept.

Figure 8 presents an example Berkeley sockets application. The socket call specifies the network and transport protocols that will be used for a connection. Thus the semantics of Berkeley sockets can include network reliability (such as with TCP/IP). What is not included is any notion of network encryption or authentication. Therefore, a program that receives a network socket file descriptor from accept does not know the remote principal associated with the connection. Likewise, connect does not authenticate the server an application connects

to. Furthermore, the specification of the application-layer protocol is absent from the Berkeley sockets API, so handling ill-formed requests is left to application code.

Labeled IPsec [124] combines IPsec and SELinux [122] to provide more comprehensive network protections. Using a domain-wide authorization policy, the system:

(1) associates SELinux labels with IPsec security associations,

(2) limits a process' security associations (connections) using a kernel authorization policy, and

(3) employs a modified inetd that executes worker processes in a security domain corresponding to the label associated with an incoming request.

In this manner, labeled IPsec can solve many of the authentication deficiencies in plain IPsec. However, labeled IPsec depends on a verified TCB and enforcement policy, even though it builds upon the Linux kernel, SELinux, and IPsec, each of which are very complex. Furthermore, IPsec security association granularity limits the granularity of controls in labeled IPsec.

An application's use of Pluggable Authentication Modules (PAM) and the setuid system call routinely provides authentication on modern Unix systems. PAM provides configurable authentication methods (e.g., Kerberos, password, smart card, etc.) using a variety of back end databases [125]. Because PAM modules typically execute in the application's address space, a compromised application may leak user secrets or bypass authentication all together. The setuid system call changes a process' user credentials, but the rules for using setuid are complex [126]. Attacks can inflict substantial damage, as setuid often requires the process to run as root

before transitioning to the target user. In addition, cryptographic algorithms are hard to get right, and in the UNIX model, updates are inconvenient. For example, if a Kerberos library needed to be updated, it would require all such applications to be relinked or at least restarted. These are the issues the Plan 9 team attempted to address in *Security in Plan 9* [98].

2.4.3 Plan 9

Plan 9 is a research operating system that started in the 1980s at Bell Labs. Many of the Plan 9 researchers were veterans of UNIX. They felt that UNIX was a good design, although it had lost some of its architectural purity after it accreted networking and graphics. Where UNIX was for time-sharing systems, Plan 9 was going to be for the new, networked workstations. Plan 9's authentication innovations took place during its fourth edition. These include factotum, a capability-based privilege transition mechanism, the secstore, and a kernel-based TLS implementation.

Factotum is a user-space dæmon and is responsible for authentication on Plan 9. The principal purpose of factotum is to move authentication code and cryptographic keys into a separate address space from an application. (The authors acknowledge that SSH performed this separation first [127], but factotum is much more general.) Factotum acts as an authentication proxy. Consider a POP email server that must implement the APOP authentication protocol. On Plan 9, such an email server would receive requests from the network and process them. In the case of authentication requests, the email server forwards the request to factotum. Factotum then provides the email server with the response it should pass to the client. Never in this process are keys shared with the email server. Factotum provides support for many authentication

protocols. Factotum addresses the leaking of keys through debugging interfaces, swap space (Plan 9 never swaps the factotum process to disk), or willing disclosure.

Plan 9 avoids the requirement for special privileges by providing a different mechanism than UNIX's setuid. Plan 9 provides two special devices, /dev/caphash and /dev/capuse. Factotum is responsible for authorizing user transitions following a successful authentication; it writes to /dev/caphash to signal to the Plan 9 kernel that a process should be allowed to transition. (Factotum is the only program that may write to /dev/caphash.) Consider a program p that wishes to transition from user u_1 to user u_2:

(1) After authenticating u_2, factotum generates a random value r and writes the string $\text{hmac}(u_1@u_2, r)$ to /dev/caphash.

(2) factotum then provides p with r.

(3) p writes $u_1@u_2@r$ to /dev/capuse.

(4) The kernel then calculates $\text{hmac}(u_1@u_2, r)$, which should match the value written by factotum. If it does, then the kernel transitions p to run as u_2.

This process by which Plan 9 performs authentication is demonstrated in detail in Figure 9. The figure shows the messages sent as time progresses from the top to the bottom of the y axis. There are four programs involved, labeled across the top: Factotum_C; Client, an email client; Server, a POP server; and Factotum_S. As indicated, the factotums act as authentication proxies, telling the client and server what authentication messages they should transmit. (Non-authentication-related POP messages are not depicted.) Towards the end of the process,

Factotum$_S$ and Server write to /dev/caphash and /dev/capuse, respectively. Upon completion, the kernel transitions the server from running as user n to user u.

Plan 9's secstore holds all of a user's keys, allowing a sort of single sign on. When a user logs in, he authenticates to factotum once with his password. Factotum connects to the user's secstore, provides its authentication credentials, and fetches the user's key file. From this point on, factotum can authenticate as the user to various services.

Plan 9 also implements a portion of the TLS protocol in kernel space. (The negotiation of the shared TLS secret is still performed in user space using factotum.) Plan 9's tlsClient procedure takes as arguments a network file descriptor and TLS connection data structure, negotiates a TLS connection, and associates this TLS connection with the kernel's TLS device. TlsClient then sets the session ID and X.509 certificate fields in the TLS connection data structure and returns an encrypted networking file descriptor to the application. At this point, the application can confirm the server's X.509 certificate and read/write encrypted messages merely by reading/writing the encrypted networking file descriptor. This continues the theme of isolating long-term cryptographic secrets from applications. On a UNIX system, libraries are often used to implement TLS, and this generally is done in such a way that adds cryptographic keys to an application's address space. Moving the TLS implementation to the kernel solves this problem.

Plan 9 provides a logical improvement to UNIX and provides techniques that could be used to increase the robustness of Kerberos implementations. However, Plan 9 applications must make explicit use of factotum. Furthermore, administrators must take care to install only

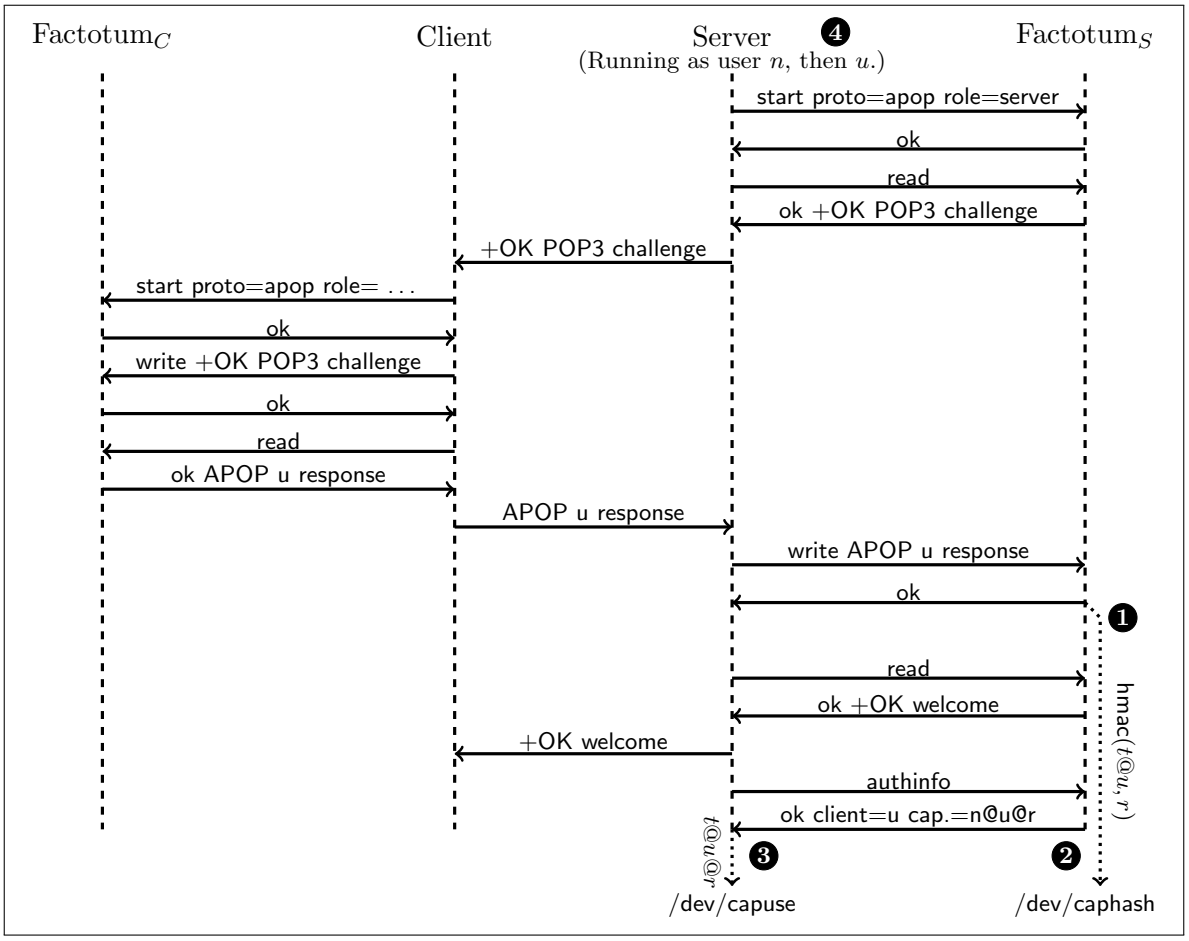


Figure 9: Plan 9 authentication: Using factotum to authenticate for an email service

these applications, and to configure them to use strong authentication and encryption. Another source of problems on Plan 9 is that TLS applications must themselves validate certificate paths, check certificate identities, and check that the certificate has not been revoked. As we discussed in 2.3.3, this has since been shown to be a routine source of programmer error. By forgoing backwards compatibility (in the sense of removing support for weak network and authentication protocols), an OS could subsume the work of developing, configuring, and auditing applications

to ensure they use Plan 9-like protections. Such a system could reduce application-based subversion, which remains in Plan 9.

2.4.4 Taos and Singularity

Taos is an OS designed at Digital Equipment Corporation to build distributed systems with strong authentication [128]. Taos implements distributed security as part of the OS, and makes no distinction between local and remote principals. IPC takes place over point-to-point channels, and each channel endpoint has a principal associated with it. A Taos process may initiate a channel to make a request on behalf of one of a set of allowed principals. The recipient can identify the requester using GetPrin and check whether the request is permitted using Check.

Taos provides the notion of *compound principals*. Compound principals can specify many constructs, such as simple principals (e.g., user *Alice* says), principals in roles (e.g., *Alice* acting as *Administrator* says), and principals quoting principals (e.g., *Bob* says *Alice* says). Such constructs empower sophisticated authorization; for example, a database might want to allow requests of the form *Webserver* says *Alice* says, while denying requests directly form *Alice*. Check's support for compound principals allows this.

Microsoft designed Singularity to demonstrate a reliable system that ran the OS kernel, device drivers, and applications within a type-safe Common Language Runtime (CLR) virtual machine. Singularity simplifies Taos' authentication model somewhat [129]; its processes have a single, immutable principal associated with them. Thus it is likely that applications will need to spawn processes for different purposes (i.e., least privilege), but Singularity's Software Isolated Processs (SIPs) lower this cost. Because isolation between SIPs is a result of programming

language type safety, all of Singularity runs in the same address space. Creating and managing SIPs is much more efficient than with traditional processes.

2.4.5 HiStar

HiStar provides a simplified, low-level interface; implements mandatory information-flow constraints; and provides a UNIX layer for compatibility [16]. DStar builds on HiStar to provide information flow across hosts on a network [130]. HiStar's flow controls contain the effect of a compromised application, serving as a countermeasure to one of the facets of application-based subversion. Even if an application is compromised, it cannot bypass the flow controls that HiStar imposes on it. However, other problems remain. An application that operates within its information-flow constraints could easily be programmed or misconfigured so that protections are missing. For example, many of the problems that arise from using TLS libraries on traditional UNIX systems, still remain with HiStar's UNIX layer.

2.4.6 Application architectures

The Multics principle of least privilege remains in several modern application architectures. These include manual privilege separation [127], automatic privilege separation partitioning [131], and implementing a user-space object manager that applies a system-wide authorization policy within an application (e.g., with SELinux) [132]. Such techniques isolate sensitive code and unify authorization policy.

Least-privilege architectures commonly require a software component we call a *distributor*. Distributors invoke a carefully-privileged process to service a request. Many existing systems use distributors, including the Internet service dæmon (inetd), the OKWS web server's okd, and

the qmail mail server's qmail-send/qmail-lspawn [133; 134]. Breaking up server applications for least privilege can stress an OS's isolation mechanisms, especially when it requires that many processes must run as many users. For example, such a server on Unix would have to execute a process per user. For this reason, OKWS on Unix provides only per-service (not per-user) isolation. Efstathopoulos et al. extended OKWS to run on the Asbestos OS and provide per-user isolation because Asbestos provides light-weight user isolation. Singularity's light-weight SIPs likewise benefit least-privilege architectures. Qmail demonstrates that per-user isolation can perform well on systems with traditional processes such as Unix.

CHAPTER 3

INTRODUCTION TO ETHOS' NETWORK API

Chapters 4–9 will describe in detail how Ethos provides network encryption, DoS protections, authentication, authorization, and type safety in a way that minimizes application-based subversion of these protections. Here we briefly introduce Ethos' network API to facilitate those discussions. As an example application, we present a line-oriented remote shell utility, resh. The function of resh is similar to SSH; a user can use resh to log into a remote Ethos computer and interact with its shell remotely. (Resh presently is limited to line-oriented interaction because it does not implement curses-like screen control [135].)

Table II summarizes the system calls that contribute to Ethos' robust networking, and Figure 10 shows their use in resh. Like POSIX, many Ethos system calls are file-descriptor-based, but Ethos' system calls provide more inherent protections [136].

Figure 10a shows the resh client. The client uses Ethos' ipc system call to make a connection to a resh server (example.com). This is the analogue of connect on POSIX, and returns a network file descriptor. It differs in that ipc's semantics specify that Ethos will:

(1) encrypt the connection (P2, Chapter 4),

(2) apply certain DoS protections (P5, Chapter 4),

(3) authenticate the resh server (P1, Chapter 5),

(4) only permit connections to authorized servers (P6, Chapter 8), and

(5) enforce an application-layer protocol (resh, in this case) on successive reads and writes (P3, Chapter 6).

Figures 10b and 10c show the server, which is split into two processes to satisfy the Multics principle of least privilege. Indeed, the authorization policy associated with resh requires such an architecture. The distributor process, reshDistributor, has the minimum privileges necessary to call advertise (the analogue of listen) and import (the analogue of accept) to receive a network connection from the client, and then to pass this connection to a per-user process. The advertise/import system calls are different than listen/accept because Ethos will:

(1) encrypt the connection (P2, Chapter 4),

(2) apply certain DoS protections (P5, Chapter 4),

(3) authenticate the remote user and provide the calling application with the user's name (P1, Chapter 5),

(4) only permit connections from authorized remote users (P6, Chapter 8), and

(5) enforce an application-layer protocol (resh, in this case) on successive reads and writes (P3, Chapter 6).

The distributor itself may not read or write the imported file descriptor. Instead, the distributor invokes reshVP, a per-user process, through the use of the fdSend system call. This call starts a new process, running as the specified user, and provides it with the network file descriptor. In the case of resh, the user provided to the fdSend call must match the remote client user. This symmetry between the per-user server and remote client user means that

Ethos can naturally restrict each per-user process. We describe fdSend in detail in Chapter 5 and authorization in Chapter 8.

What should be striking in this example is the lack of encryption, authentication, and authorization calls in the application. Ethos centralizes such protections in the OS and therefore makes them compulsory. This means that an application cannot avoid them, whether by programmer omission, misconfiguration, or application compromise. Indeed, Ethos authenticates all network requests, and a remote client user may not interact with server application code at all unless authorized by Ethos. Furthermore, Ethos submits each network request to an RPC type checker, so an Ethos application never receives ill-formed application-layer data from the network. Thus applications only receive network requests that are authorized and well-formed.

Moving protections into the OS gives rise to another advantage: Ethos carefully isolates secret cryptographic keys within its kernel, never releasing them to applications (P4). Instead, applications indirectly make use of keys through the use of system calls, including authenticate, sign, and the network system calls. On Ethos, application-based subversion cannot directly leak secret keys.

Isolating keys in the kernel also means that Ethos can enforce the use of particular cryptographic implementations for its protections. For example, there is no need to inspect each application to ensure the use of side-channel-resistant cryptographic implementations. Furthermore, Ethos can zero private keys after use in many instances to reduce the effectiveness of a cold boot attack; unlike many systems, Ethos' isolation of private keys in kernel space means

```
1  func reshExecCommandReply (e *Encoder, response string)
2      print (response)

4  // Main.
5  e, d ← resh.Ipc ("resh", "example.com")
6  do forever
7      command ← readCommand ()
8      e.ReshExecCommand (command)
9      d.HandleResh (e)
```

(a) Client resh connects, issues requests, and receives responses

```
10 listenFd ← advertise ("resh")
11 do forever
12     netFd, user ← import (listenFd)
13     fdSend ([netFd], user, "reshVP")
```

(b) Server reshDistributor sends connections to reshVP

```
14 func processCommand (command)
15     // Not shown: pipe, fork, execute command, and so on.

17 func reshExecCommand (e *Encoder, command []string)
18     if command = "exit"
19         exit ()
20     response ← processCommand (command)
21     e.ReshExecCommandReply (response)

23 // Main.
24 fd ← fdReceive ()
25 e ← etnEthos.NewWriter (fd)
26 d ← etnEthos.NewReader (fd)
27 do forever
28     d.HandleResh (e)
```

(c) Per-user reshVP services shell requests

Figure 10: The Ethos remote shell application resh in pseudo code

that such sensitive memory locations are known by the kernel (i.e., not lost due to the semantic gap between the kernel and an application).

TABLE II: The system calls that contribute to Ethos' robust networking

Ethos call	Analogue	Ch.	Key Ethos semantics
Process			Processes have an immutable user
fork	fork	5	
exec	exec	5	
exit	exit	5	
Networking			
advertise	listen	5, 8	Services are named by strings, not integer ports
			Each service has a corresponding filesystem node
import	accept	5, 8	Ethos encrypts all networking
			Ethos applies certain DoS protections
			Ethos cryptographically authenticates the remote user
			Ethos permits only connections from authorized users
			Ethos isolates cryptographic keys from applications
ipc	connect	5, 8	Ethos encrypts all networking
			Ethos applies certain DoS protections
			Ethos cryptographically authenticates the server
			Ethos permits only connections to authorized servers
			Ethos isolates cryptographic keys from applications
Authentication			
authenticate	n/a	5	Ethos isolates passwords from applications
			Does not require special privileges
fdSend	sendmsg	5, 8	Invokes a virtual process, run as the specified user
fdReceive	recvmsg	5, 8	
I/O			
read	read	5, 6, 8	Subjected to Ethos' type checker
write	write	5, 6, 8	—"—
peek	n/a	5, 6, 8	—"—
Miscellaneous			
createDirectory	mkdir	6	Directories bear labels and types which apply to files

CHAPTER 4

MINIMAL LATENCY TUNNELING

Our goal with Ethos is to protect *all* networking against eavesdropping, modification, and, to the extent possible, DoS. To achieve this goal, Ethos networking must identify users and servers using strong authentication, ensure header and data integrity, provide DoS protections, provide confidentiality, and disrupt traffic-flow analysis. In addition, the protocol must perform well and minimally burden application programmers and administrators. These needs are not met by existing protocols.

We already described how hardware and software improvements have eliminated historical cryptographic performance bottlenecks, but one performance parameter has a fundamental limitation—network latency [137]. Latency is critical for users [138]. For example, Google found that a 500ms latency increase resulted in a 25% dropoff in page searches, and older studies have shown that user experience degrades when interfaces operate with latencies as small as 100ms [139]. We describe here MinimaLT, a network protocol which delivers protected data on the first packet of a typical client-server communication, provides substantial protections, and is extraordinarily simple to configure and use. MinimaLT is Ethos' native network protocol.

Particularly challenging has been to provide PFS at low latency. Traditionally, using DH to achieve PFS requires a round trip before sending any sensitive data. MinimaLT eliminates this roundtrip, instead receiving the server's ephemeral key during a directory service lookup (§4.1.3.1). To establish connection liveness—necessary only if the connection is inactive and the

server is running out of memory—we invert the normal mandatory start-of-connection handshake and replace it with an only-when-needed server-originated puzzle handshake (§4.1.3.4).

A second challenge is to make connections portable across IP addresses to better support mobile computing. MinimaLT allows you to start a connection from home, travel to work, and continue to use that connection. This avoids application recovery overhead and lost work for operations which would otherwise be interrupted by a move. MinimaLT IP mobility does not require intermediary hosts or redirects, allowing it to integrate cleanly into protocol processing (§4.1.3.3). To provide better privacy, MinimaLT blinds third parties to IP-address changes, preventing them from linking a connection across different IP addresses.

Other challenges include DoS, authentication, and authorization. MinimaLT dynamically increases the ratio of client (i.e., attacker) to server resources needed for a successful DoS attack, deploying a variety defenses to maintain proportional resource usage between a client and server (§4.2.3). We also designed MinimaLT to integrate into systems with strong authentication and authorization, and its authentication framework supports identified users, strangers, and anonymous users (§4.1.1.1).

To meet these challenges, we have applied Ethos' clean-slate strategy to design MinimaLT, starting from User Datagram Protocol (UDP), and intentionally crossing multiple network layers. This approach has two substantial benefits. First, security problems often occur in the seams between layers. For example, TLS is unable to protect against attacks on TCP/IP headers due to its layering: RST and sequence number attacks interrupt TLS connections in

such a way that is difficult to correct or even detect [140; 141; 142]. Second, multi-layer design enables MinimaLT to achieve higher performance.

During our design process, we found an unexpected synergy between speed and security. The Internet uses higher-latency protocols because, historically, low-latency, stream-oriented protocols such as T/TCP have allowed such severe attacks [143] as to make them undeployable. It turns out that providing strong authentication elsewhere in the protocol stops all such attacks without adding latency. Eliminating roundtrips makes MinimaLT *faster than cleartext TCP/IP at establishing connections.*

In short, MinimaLT provides the features of TCP/IP (reliability, flow control, and congestion control), and adds in encryption, authentication, clean IP mobility, and DoS protections, all while preserving PFS and reducing latency costs. The design of MinimaLT provides encryption and authentication with PFS while allowing a client to include data in the first packet sent to a server (often forgoing pre-transmission round trips entirely). Table III compares several of the network protocols we introduced in Chapter 2 to MinimaLT. In the rest of this chapter, we describe MinimaLT and its implementation.

4.1 Design

4.1.1 Overview

We begin with a high-level introduction of the MinimaLT protocol design. We co-designed MinimaLT along with Ethos, and MinimaLT serves as Ethos' native networking protocol. Co-design ensures that the MinimaLT protocol integrates well with Ethos authentication, authorization, and naming. MinimaLT identifies hosts using public-key cryptography, multiplexes

TABLE III: Summary of network protocols

	TCP /IP	TCP Fast Open	SST	IPsec	Labeled IPsec	TLS	False Start	Snap Start	Tcpcrypt	MinimaLT
Encryption (P2)			✓	✓	✓	✓	✓	✓	✓	✓
PFS				✓	✓	✓	✓	✓	✓	✓
User authentication (P1)					✓	✓	✓	✓	✓	✓
Protection of sequence numbers				✓	✓					✓
Robust DoS protections (P5)										✓
IP-address mobility with protections against linking										✓
Only strong cryptography supported (P2)										✓
Round trips before client sends data*	2	2	2	≥4	≥4	4	3	2	3	1
. . . if server is already known	1	1	1	≥3	≥3	3	2	1	2	0
. . . in abbreviated case†	1	0	0	1	1	2	2	1	1	0

*Includes one round trip for DNS/directory service lookup of unknown server

†Assumes protocol-specific information cached from previous connection to same server

multiple authenticated user-to-service connections within a single encrypted host-to-host tunnel, lowers latency by replacing setup handshakes with the publication of ephemeral keys in a directory service, and builds on carefully designed cryptographic abstractions. We have also ported MinimaLT to Linux; even in this environment, MinimaLT cleanly provides many security and performance advantages.

4.1.1.1 Public key

MinimaLT is decidedly public-key-based. Both servers and users are identified by their public keys; such keys serve as a UUID [128; 95; 144]. Principals prove their identity by providing ciphertext which depends on both their and the server's keys. A principal may be

known—i.e., the underlying OS is aware of a real-world identity associated with the principal's public key—or he may be a stranger or anonymous.

Whether strangers are allowed is left to the underlying OS's authorization policy. In Ethos, a server can be authorized to create sparse stranger accounts on demand. That is, once a server encounters a stranger, the server may isolate the stranger's resources from other users and they remain available if he returns. We discuss Ethos authorization in Chapter 8.

4.1.1.2 Network tunnel

A MinimaLT *tunnel* is a point-to-point entity that encapsulates the set of connections between two hosts. MinimaLT creates a tunnel on demand in response to the first packet received from a host or a local application's outgoing connection request. Tunnels provide server authentication, encryption, congestion control, and reliability; unlike with TLS/TCP, these services are not repeated for each individual connection.

Tunnels make it more difficult for an attacker to use traffic analysis to infer information about communicating parties. Of course, traffic analysis countermeasures have limits [145]; for obvious cost reasons we did not include extreme protections against traffic analysis, such as using white noise to maintain a constant transmission rate.

A MinimaLT server does not respond unless the remote user is authorized to connect. In existing protocol layering, this is possible only using weak, IP-address-based authentication; as we show, MinimaLT authentication is much stronger. Thus MinimaLT makes network mapping much more difficult. Since most hosts do not offer Internet-wide services, they present a minimal signature to attackers—even the MinimaLT equivalent to ping will respond only to

authorized users. Alternatively, MinimaLT servers might be configured to provide services to anonymous users. This is appropriate for services intended for the Internet at large.

As with IPsec, a MinimaLT tunnel provides cryptographic properties that ensure confidentiality and integrity. However, MinimaLT has been designed to provide tighter guarantees than IPsec, which (as generally used in practice) provides host-based protections only. In MinimaLT, connections provide user authentication as described below.

4.1.1.3 Connections

A MinimaLT tunnel contains a set of *connections*, that is, a single tunnel between two hosts encapsulates an arbitrary number of connections. Each connection is user-authenticated and provides two-way communication between a client application and service. In addition to multiplexing any number of standard application-to-service connections, each MinimaLT tunnel has a single *control connection*, along which administrative requests flow (§4.1.2.3).

4.1.1.4 Directory service

Central to our protocol is the MinimaLT *directory service*. The directory service resolves queries for hostname information, providing a server's directory certificate, signed by the server's long-term key. This returned certificate contains the server's IP address, UDP port, long-term key, zero padding (the minimum payload size of the initial packet), and an ephemeral key. An additional certificate vouches for the server's long-term public key and ties it with the server's hostname. Servers register with a MinimaLT directory service to provide this information, and they update their ephemeral key at a rate depending on their security requirements. In §4.1.4 we describe how directory services integrate with DNS to span the Internet.

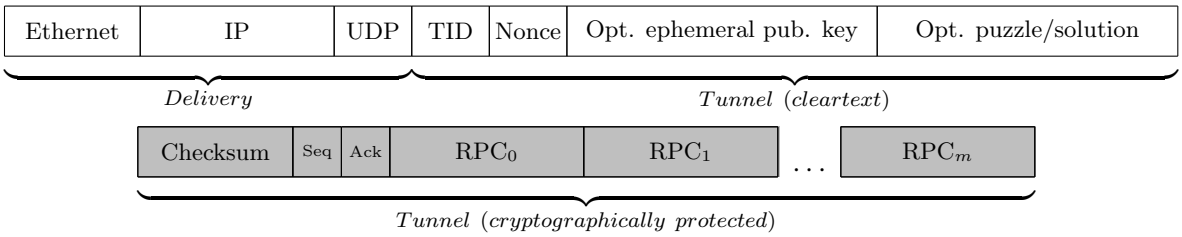
4.1.1.5 Cryptography

MinimaLT uses NaCl, a cryptographic suite which includes symmetric encryption algorithms, cryptographic hashing, DH, and signatures [146; 147]. We chose NaCl for its high performance, simple interface, and immunity to software side-channel attacks. NaCl's symmetric cryptography provides both encryption and integrity protection. MinimaLT also uses NaCl's ECC DH and SHA-512 cryptographic hashing.

NaCl's box encrypts and authenticates, and box_open verifies and decrypts. These routines come in two flavors, a symmetric key version which takes a single key and a public key version which takes one party's public key and the other party's private key. In addition to the key(s), box/box_open take plaintext/ciphertext and a number-used-once (nonce).

4.1.2 MinimaLT layering and packet format

MinimaLT can be broken into three conceptual layers: delivery, tunnel, and connection, and these layers correspond to the design described above. The *delivery layer* contains routing and other information necessary to deliver a packet to its destination host. This layer is made up of the standard Ethernet, IP, and UDP protocols. The *tunnel layer* provides the cryptographic and reliability information necessary to support MinimaLT tunnels. This includes server authentication, encryption, cryptographic integrity protections, DoS protections, and congestion control. Finally, the *connection layer* provides for application-to-application RPCs. Here we introduce these layers and their fields before describing in more detail how the MinimaLT protocol uses them in §4.1.3.



(a) MinimaLT packet format with cleartext in white and cryptographically protected portion in gray

	Size (bytes)	
Field	**First**	**Successive**
Delivery fields		
Ethernet header	14	14
IP	20	20
UDP	8	8
Cryptographic fields		
Tunnel ID	8	8
Nonce	8	8
Ephemeral public key	32	n/a
Puzzle/solution	148	n/a
Checksum	16	16
Reliability fields		
Sequence num.	4	4
Acknowledgment	4	4
Connection fields		
Connection ID	4	4
RPC	variable	
Total (except RPC)	282	86

(b) Header field sizes in a MinimaLT tunnel's first/successive packets

Figure 11: MinimaLT packet format and header field sizes

The packet format itself is simple and is given in Figure 11. The cleartext portion of the packet contains the Ethernet, IP, and UDP headers; the Tunnel ID (TID); a nonce; and two optional fields used at tunnel initiation—an ephemeral public key and a puzzle. A client provides the public key only on the first packet for a tunnel, and a server requires puzzles opportunistically to prevent memory and computation attacks.

The packet's cryptographically protected portion contains ciphertext and the ciphertext's cryptographic checksum. The ciphertext contains a sequence number, acknowledgment number, and a series of RPCs. In contrast to byte-oriented protocols, we use RPCs at this layer because they result in a clean design and implementation; they are also general and fast (§4.2). RPCs have a long history dating to the mid-1970s [148; 149].

4.1.2.1 Delivery layer

The usual Ethernet, IP, and UDP headers allow the delivery of packets across existing network infrastructure; they play no role in any packet processing within MinimaLT. The UDP header allows packets to traverse NATed networks [150], and it enables user-space implementations of MinimaLT. Aside from the length field, the UDP fields are otherwise uninteresting for MinimaLT.

4.1.2.2 Tunnel layer (cryptography and reliability)

The tunnel establishment packet (the first packet sent between two hosts) contains a TID, a nonce, and the sending host's (ephemeral) public DH key. The TID is pseudo-randomly generated by the initiator. The public key is ephemeral to avoid identifying the client host to a third party, but onion routing might be necessary to also blind the IP address [35].

The recipient cryptographically combines the client's ephemeral public key with its own ephemeral private key to generate the symmetric key (§4.1.3). The recipient then uses this symmetric key and the nonce to verify and decrypt the encrypted portion of the packet.

After tunnel establishment, the tunnel is identified by its TID. Successive packets embed the TID, which the recipient uses to look up the symmetric key which in turn is used to decrypt the payload. Thus MinimaLT reduces packet overhead by using TIDs rather than resending the public key. The TID is 64 bits—one quarter the size of a public key—with one bit indicating the presence of a public key in the packet, one bit indicating the presence of a puzzle/solution, and 62 bits identifying a tunnel.

The nonce ensures that each payload transmitted between two hosts is uniquely encrypted. The nonce is a monotonically increasing value; once used, it is never repeated for that (unordered) pair of keys. The nonce is odd in one direction (from the side with the smaller key) and even in the other direction, so there is no risk of the two sides generating the same nonce. Clients enforce key uniqueness by randomly generating a new ephemeral public key for each new tunnel; this is a low-cost operation. For a host which operates as both client and server, its client ephemeral key is in addition to (and different from) its server ephemeral key.

The tunnel layer also contains an optional field that might contain a puzzle request or solution (§4.1.3.1). The purpose of the puzzle here is to protect against spoofed tunnel requests. Such an attack might cause a server to perform relatively expensive DH computations. A puzzle both demonstrates connectivity and expends initiator resources, and thus limits attack rates. (In addition to the puzzle header fields, MinimaLT provides puzzle RPCs to defend against

Sybil attacks [151; 61] after a tunnel has been established. We describe the details of both in §4.1.3.4 and evaluate them in §4.2.3.)

Beyond the cleartext fields mentioned so far, the tunnel layer contains a strong 128-bit cryptographic checksum (a keyed message-authentication code) of the ciphertext. An attacker does not have the shared symmetric key, so any attempt to fabricate ciphertext will be detected.

The final component of the tunnel header consists of reliability information in the form of sequence and acknowledgment fields. MinimaLT's reliability and connection (described below) headers are part of the ciphertext, so they are protected against tampering and eavesdropping. The total size of the delivery, reliability, and connection headers is equal to that of the headers in TCP/IP. We discuss packet overhead further in §4.2.1.

4.1.2.3 Connection layer

MinimaLT's connection layer supports an arbitrary number of connections, where each connection hosts a series of RPCs. We depict RPCs using the form $f_c(a_0, a_1, \ldots)$, where f is the name of the remote function, c is the connection that the RPC is sent to, and $a_0, a_1, \ldots$ are its arguments. A single packet can contain multiple RPCs, which amortizes the overhead due to MinimaLT's delivery and tunnel fields across multiple connections. We discuss Ethos' particular RPC encoding format in Chapter 6.

One connection is distinguished: connection 0 is the control connection and hosts the management operations which maintain the tunnel and its other connections. These control RPCs create, close, accept, and refuse connections; provide certificates (§4.1.3.1); perform rekeying

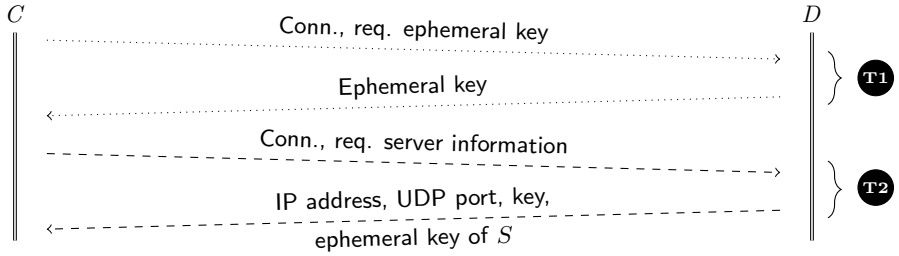
(§4.1.3.2); support IP address changes (§4.1.3.3); support puzzles (§4.1.3.4); authenticate users (§4.1.3.6); and support window size adjustments (§4.1.3.7):

RPC	**Description**
$\mathsf{create}_0(c,\ s)$	create an anonymous connection c of type s
$\mathsf{createAuth}_0(c,\ s,\ U,\ x)$	create an authenticated connection for the user with long-term public key U, who generates authenticator x
$\mathsf{close}_0(c)$	close connection c
$\mathsf{ack}_0(c)$	creation of c successful
$\mathsf{refuse}_0(c)$	connection c refused
$\mathsf{requestCert}_0(H)$	get host H's certificate
$\mathsf{provideCert}_0(\mathsf{cert})$	provide the certificate *cert*
$\mathsf{ok}_0()$	last request was OK
$\mathsf{nextTid}_0(t, C')$	advertise future TID to prepare for a rekey or IP address change
$\mathsf{puzzle}_0(p,\ h,\ w)$	pose a puzzle
$\mathsf{puzzleSoln}_0(p,\ h,\ w)$	provide a puzzle solution
$\mathsf{windowSize}_0(c,\ n)$	adjust connection receive window

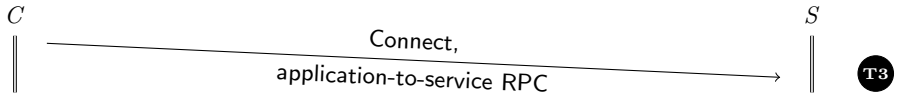
In general, each service provided by a host supports a set of service-specific RPCs on standard connections, so data on these connections are sent unchanged to their corresponding applications. Our illustrations use the following sample RPC:

$\mathsf{serviceRequest}_c(...)$	a request for some type of service on connection c

The tunnel, and the RPCs within it, are totally sequenced. Thus RPCs are executed in order (as opposed to separately implemented connections—as in TLS—where ordering between connections is not fixed). This enables a clean separation of the control connection from other connections. We have found that this simplifies both the protocol and its implementation.



(a) Prelude to connection establishment: performed if the tunnel does not yet exist



(b) Connection establishment

Figure 12: MinimaLT protocol trace

4.1.3 Protocol

The purpose of the protocol is to allow protected communication between a client and server. We discuss here the establishment of a symmetric key §4.1.3.1, rekeying §4.1.3.2, IP-address mobility §4.1.3.3, puzzles §4.1.3.4, the absence of a three-way handshake §4.1.3.5, user authenticators §4.1.3.6, and congestion control §4.1.3.7.

4.1.3.1 Establishing the symmetric key

MinimaLT approaches PFS, encrypting sensitive data using only ephemeral keys; it protects both client-side data and identity. In this section, we will show the negotiation of keys and the transmission of RPCs in the absence of puzzles. This is the normal case, when servers are not under heavy load.

At least three entities cooperate to establish a symmetric key: a client C, a directory service D, and a server S. Here we discuss the intra-organizational case, where a single organization

maintains C, D, and S. In §4.1.4 we show how MinimaLT scales to the Internet using DNS, while providing security at least as strong as DNSSEC [152] with no additional latency. Eventually, a pure MinimaLT solution could span the entire Internet.

The client can compute the shared secret after a maximum of two round trips, and can include application data in the first packet sent to the server. (The common case is that the client can immediately compute the shared secret with zero round trips.) Each trip uses a different tunnel:

T1 C establishes a tunnel, anonymously, to D in order to obtain D's ephemeral public key;

T2 C establishes a tunnel to D using ephemeral keys to lookup S's contact information; and

T3 C establishes a tunnel to S using ephemeral keys.

Figure 12 depicts this process. C establishes tunnel T1 once, at boot time. This is the only tunnel that does not use a server ephemeral key, so C does not yet provide D with a user authenticator. Next, C establishes tunnel T2 to collect the information necessary for the first connection between C and S. It uses this information to establish tunnel T3. The tunnel establishment packet for tunnel T3 may include application-to-service RPCs. Successive connections to S skip T1 and T2, and tunnel T2 remains open to look up other servers. We use the following notation to describe the details:

---

t	A tunnel ID (described in §4.1.2)

n	A nonce (described in §4.1.2)
s	A sequence number
a	An acknowledgment number
0 or c	The connection ID
z	A puzzle
z'	A puzzle solution
C, D, S	The client, directory, and server long-term public/private key
C', D', S'	An ephemeral client, directory, and server public/private key
$C \to S$	A message from the client to the server, using keys C and S
$H(m)$	The cryptographic hash of message m
$\boxed{m}\ {}^{k}_{n}$	Encrypt and authenticate message m using symmetric key k and nonce n
$\boxed{m}\ {}^{S \to P}_{n}$	Encrypt and authenticate message m using a symmetric key derived from public key P and private key S; n is a nonce

We show each packet on a single line such as

$$t, n, C', \boxed{s, a, \ldots}\ {}^{C' \to S'}_{n}$$

which indicates a tunnel establishment packet (due to the presence of the cleartext C', as described in §4.1.2.2) from C to S using keys C' and S' to box (encrypt and authenticate) the message '$s, a, \ldots$'. Each packet has a new nonce but for conciseness we simply write n rather than n_1, n_2, etc. The same comment applies for sequence numbers (s) and acknowledgments (a).

At boot time, C establishes a tunnel with D (Figure 12a, tunnel T1); C's configuration contains D's IP address, UDP port, and long-term public key D. First, C generates a single-use public key C' and uses it to create a bootstrap tunnel with the directory service.

$$t, n, C', \boxed{s, a, \mathsf{requestCert}_0(D)} \quad {}^{C' \to D}_{n}$$

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Dcert})} \quad {}^{D \to C'}_{n}$$

D responds with a certificate containing its own ephemeral key, D', signed by D. C uses this to establish a PFS tunnel to request S's directory certificate. Tunnel T2 uses a fresh C' and is established by:

$$t, n, C', \boxed{s, a, \mathsf{requestCert}_0(S)} \quad {}^{C' \to D'}_{n}$$

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Scert})} \quad {}^{D' \to C'}_{n}$$

After receiving S's Scert, C is ready to negotiate tunnel T3. C encrypts packets to the server using S' (from Scert) and a fresh C'. Because C places its ephemeral public key in the first packet, both C and S can immediately generate a shared symmetric key using DH without compromising PFS. Thus C can include application-to-service data in the first packet. That is,

$$t, n, C', \boxed{s, a, \mathsf{nextTid}_0(t, C'), \mathsf{createAuth}_0(1, serviceName, \mathrm{U}, \mathrm{x}), \mathsf{serviceRequest}_1(\ldots)} \quad {}^{C' \to S'}_{n}$$

We describe the purpose of $\mathsf{nextTid}_0$ in §4.1.3.2. Upon receiving $\mathsf{createAuth}_0$, S verifies the authenticator x (§4.1.3.6) and decides if the client user U is authorized to connect. If so, S creates the new connection (with ID 1). The server ensures no two tunnels share the same C'. The service-specific $\mathsf{serviceRequest}_1$ can then be processed immediately on the new connection.

Tunnels are independent of the IP address of C; this means that C can resume a tunnel after moving to a new location (typically prompting the use of the next ephemeral key as described below), avoiding tunnel-establishment latency and application-level recovery due to a failed connection. This reduced latency is useful for mobile applications, in which IP-address assignments may be short lived, and thus overhead may prevent any useful work from being done before an address is lost.

Before a client may connect, S must register its own IP address, UDP port, public key, and current ephemeral public key with D. This is done using the $\mathsf{provideCert}_0$ RPC (here S has already established a tunnel with D as we describe above):

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Scert})} \quad {}^{S' \to D'}_{n}$$

$$t, n, \boxed{s, a, \mathsf{ok}_0()} \quad {}^{D' \to S'}_{n}$$

4.1.3.2 Rekey

PFS requires periodic rekeying, so that old encryption keys can be forgotten. This prevents an attacker who compromises a host from decrypting the packets encrypted using previous keys. Prior to a rekey, a host creates the next TID t and sends it to the opposite host using $\mathsf{nextTid}_0$.

(We describe the C' argument to $\mathsf{nextTid}_0$ below.) Although either side may invoke $\mathsf{nextTid}_0$, rekeying is initiated only by the client.

We reduce the rekeying workload for the client and server as follows: To rekey, the client sends a tunnel initiation packet using the next TID. The client generates a one-time valid key pair used for this initiation and places the public part of this key pair in this packet so that it is indistinguishable from a true tunnel initiation packet. (For obvious reasons, the client-generated public key is never reused for a successive rekey. Instead, future keys are likewise randomly selected.) However, instead of computing a new shared secret using DH, the client simply uses the hash of the previous symmetric key. The client then repeats its one-time public key inside the boxed part of the message (i.e., as the C' argument to another $\mathsf{nextTid}_0$).

When the server receives a packet whose TID matches a known next TID, the server hashes the existing symmetric key for that tunnel to produce the new key, and then verifies and decrypts the packet. The server also verifies that the public key sent in clear matches the public key inside the boxed part of the message. (Without this verification, a MitM attacker could modify the public key sent in the clear, observe that the server still accepts this packet, and confidently conclude that this is a rekey rather than a new tunnel.) If both verifications succeed then the server updates the tunnel's TID and handles the packet; otherwise it behaves as for failed tunnel initialization. Thus the rekey process inflicts neither superfluous round trips nor server public-key operations.

Typically, clients invoke $\mathsf{nextTid}_0$ immediately after creating a new tunnel, and after assuming a new TID/key. Servers invoke $\mathsf{nextTid}_0$ if their PFS interval expires. Clients then assume

a new TID/key when their rekey interval expires or immediately after receiving a $\mathsf{nextTid}_0$ from the server (the latter implies that the server's key has expired).

A client-side administrator sets his host's rekey interval as a matter of policy. The server's policy is slightly more sensitive, because the server must maintain its ephemeral key pairs as long as they are advertised through the directory service. An attacker who seizes a server could combine the ephemeral keys with captured packets to regenerate any symmetric key within the ephemeral key window. Thus even if the client causes a rekey, the server's ephemeral key window dominates on the server side. This asymmetry reflects reality, because each side is responsible for their own physical security. The client knows server policies and can restrict communication to acceptable servers.

4.1.3.3 IP-address mobility

Because MinimaLT identifies a tunnel by its TID, the tunnel's IP and UDP port can change without affecting communication; indeed, one purpose of $\mathsf{nextTid}_0$ is to support IP-address mobility. After changing its IP address or UDP port, a host simply assumes the next TID as with a rekey. The other host will recognize the new TID and will transition the tunnel to the new key, IP address, and UDP port. Thus a computer can be suspended at home and then brought up at work; an application which was in the middle of something could continue without any recovery actions.

MinimaLT reduces an attacker's ability to link tunnels across IP address changes because its TID changes when its IP address changes. What remains is temporal analysis, where an attacker notices that communication on one IP address stops at the same time that communication on

another starts. However, the attacker cannot differentiate for sure between IP mobility and an unrelated tunnel establishment. Blinding information below the network layer—for example, the Ethernet MAC—is left to other techniques.

4.1.3.4 Puzzles

MinimaLT uses puzzles selectively, so that their costs are only incurred when the server is under load. There are two ways MinimaLT can pose puzzles: as a part of the tunnel header (to avoid abusive DH computations) or by using the puzzle RPCs after establishing a tunnel. In the former case, a MinimaLT server under load that receives a tunnel establishment packet from a stranger for a stranger-authorized service does not create a tunnel. Instead, it responds with a puzzle:

$$z' = \boxed{C', S'}\ {}^{k}_{n'}$$

where k is a secret known only to the server. It then calculates z'' by zeroing z''s rightmost w bits (i.e., the client will take $O(2^{w-1})$ operations to solve the puzzle), where the server dynamically selects w based on its policy. The server sends the puzzle $z = [z'', H(z'), w, n']$ to the client:

$$t, n, z$$

The server forgets about the client's request. The client must solve the puzzle z and provide the solution z' along with n' in a new tunnel establishment packet using the same C' and S'.

The client brute forces the rightmost w bits of z'' to find z' with a matching hash and responds to the server with:

$$t, n, [z', n'], \boxed{s, a, \ldots}\ {}^{C' \to S'}_{n}$$

To confirm a solution, the server decrypts z' using k and n', confirms C' and S' and ensures that n' is within an acceptable window. Although the server has forgotten z', these protections ensure that the puzzle cannot be reused for other tunnel establishment attempts.

After a tunnel has been established, hosts can use the puzzle RPCs to perform small-w proof-of-life/liveness challenges on idle tunnels that might befit garbage collection. Stranger-authorized servers can also use the puzzle RPCs to slow Sybil attacks, whereby an attacker tries to generate many identities to cause public-key authenticator validations on the server. We evaluate MinimaLT's puzzles in §4.2.3.

4.1.3.5 No transport-layer three-way handshake

As described above, MinimaLT establishes an ephemeral symmetric key with a minimal number of round trips; MinimaLT also subsumes the need for a transport-layer three-way handshake when establishing each application-to-service connection. TCP's three-way handshake establishes a random ISN. This is necessary for two reasons: (1) the ISN serves as a weak authenticator (and liveness check) because a non-MitM attacker must predict it to produce counterfeit packets, and (2) the ISN reduces the likelihood that a late packet will be delivered to the wrong application.

MinimaLT encrypts the sequence number, provides cryptographic authentication, and checks liveness using puzzles, addressing (1). MinimaLT uses TIDs, connection IDs, and nonces to detect late packets, addressing (2). Thus MinimaLT can include application data in a connection's first packet, as discussed above. Extra round trips are necessary only if the tunnel does not exist; and then only when the client does not have S's directory certificate or is presented with a puzzle. If the server provides a puzzle, it means that the server is under heavy load so that additional latency is unavoidable.

4.1.3.6 User authenticators

Every user serviced by MinimaLT is identified by his public key. The $\mathsf{createAuth}_0$ authenticator is the server's long-term public key encrypted and authenticated using the server's long-term public key, the user's long-term private key U, and a fresh nonce n:

$$x = \boxed{S}\ {}^{U \to S}_{n}$$

Because authenticators are transmitted inside boxes (as ciphertext), they are protected from eavesdropping, and because the authenticator is tied to the server's public key, the server cannot use it to masquerade as the user to a third-party MinimaLT host. Of course, any server could choose to ignore the authenticator or perform operations the client did not request, but that is unavoidable. If third-party auditability is desired then users can choose to interact only with servers that take requests in the form of certificates.

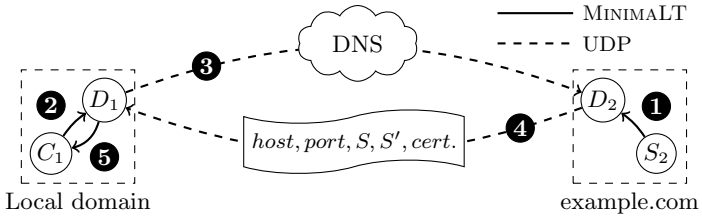


Figure 13: An external directory service query

4.1.3.7 Congestion/Flow control

MinimaLT's tunnel headers contain the fields necessary to implement congestion control, namely sequence number and acknowledgment fields. We presently use a variation of TCP's standard algorithms [153]. As with TCP [154], efficient congestion control is an area of open research [155], and we could substitute an emerging algorithm with better performance. MinimaLT does have one considerable effect on congestion control: controls are aggregated for all connections in a tunnel, rather than on individual connections. Since a single packet can contain data for several connections, the server no longer needs to allocate separate storage for tracking the reliability of each connection. This also means that MinimaLT need not repeat the discovery of the appropriate transmission rate for each new connection, and a host has more information (i.e., multiple connections) from which to derive an appropriate rate. The disadvantage is that a lost packet can affect all connections in aggregate.

MinimaLT hosts adjust per-connection flow control using the $\mathsf{windowSize}_0$ RPC. MinimaLT subjects individual connections to flow control, so $\mathsf{windowSize}_0$ takes as parameters both a connection ID and size. MinimaLT currently implements TCP-style flow control.

4.1.4 A directory service that spans the Internet

Within an organization, an administrator maintains clients, servers, and a directory service. However, clients will often want to connect to services outside of their organization, so it becomes necessary to obtain external servers' directory service records. MinimaLT integrates disparate directory services using DNS in a way that does not add latency to the current requirement of performing a DNS lookup. MinimaLT directory services support their organization as described above, but also can make DNS queries about external hosts and can service DNS queries about local hosts. (We note that this section describes how to integrate MinimaLT with the existing Internet infrastructure while providing at least the security provided by DNS/TLS; we would prefer to replace DNS with a pure MinimaLT infrastructure because this would provide stronger security.)

We depict an external lookup in Figure 13. As described in §4.1.3.1, servers such as S_2 publish their ephemeral keys to their local directory service (1). To connect to an external server, e.g., example.com's S_2, the client C_1 requests S_2's information from C_1's directory service D_1 (2), and, if cached, D_1 immediately replies. Otherwise, D_1 makes a DNS request for example.com's S_2 (3). The DNS reply from example.com's DNS/directory service D_2 is extended to contain a full MinimaLT server record, split into one long-term DNS record containing S_2's long-term key and a chain certifying the identity of this key, and one shorter-term DNS record containing an IP address, a UDP port, and an ephemeral server key, all signed by S_2. Once D_1 receives this DNS reply (4), it can respond to C_1's request as earlier described (5).

The integration of MinimaLT's directory services with DNS affects DNS configurations in two ways. First, the shorter-term DNS record's time to live must be set to less than or equal to the rekey interval of the host it describes. We expect this to have a light impact, because many organizations already select short times to live. Second, DNS replies will grow due to the presence of additional fields. The largest impact is the identity certificate, which we expect will initially be encoded as X.509, although smaller certificate encodings could reduce this cost. We also expect to initially adopt a certificate-authority-based trust model, because this is most commonly used on the Internet. Of course, future revisions could adopt stronger and more flexible trust models, and any latency impact they might incur would also be present if they were applied to DNS/TLS.

4.2 Evaluation

We summarized the security protections and other key properties of MinimaLT in Table III. Here we evaluate MinimaLT's packet overhead (§4.2.1); the performance of creating new tunnels, creating connections on existing tunnels, and transmitting data (§4.2.2); DoS defenses (§4.2.3); and prospects for further performance improvements (§4.2.4).

4.2.1 Packet header overhead

MinimaLT's network bandwidth overhead is modest. A small additional overhead when compared to TCP/IP is due to MinimaLT's cryptographic protections, and includes the nonce, TID, and checksum (the public key/puzzle fields are rarely present, so they are insignificant overall). Thus MinimaLT requires 32 bytes more for its headers than TCP/IP; this represents 6% of the minimum Internet MTU of 576 bytes, and 2% of 1518-byte Ethernet packets.

4.2.2 Performance evaluation

We experimentally evaluate MinimaLT's performance in three areas:

(1) the serial rate at which MinimaLT establishes tunnels/connections, primarily to study the effect of latency on the protocol;

(2) the rate at which MinimaLT establishes tunnels/connections when servicing many clients in parallel; and

(3) the throughput achieved by MinimaLT.

All of our performance tests were run on two identical computers with a 4.3 GHz AMD FX-4170 quad-core processor, 16GB of memory, and a Gb/s Ethernet adapter. We benchmarked in 64-bit mode and on only one core to simplify cross-platform comparisons.

4.2.2.1 Serial tunnel/connection establishment latency

In each of our serial connection benchmarks, a client sequentially connects to a server, sends a 28-byte application-layer request, and receives a 58-byte response. We measure the number of such operations completed in 30 seconds, where each measurement avoids a DNS/directory service lookup. We performed this experiment under a variety of network latencies using Linux's netem interface.

We compare against OpenSSL 1.0.0j using its s_server and s_time utilities. We first configured an OpenSSL client and server to use 2,048-bit RSA as recommended by the US National Institute of Standards and Technology (NIST) [156], along with 128-bit AES, ephemeral DH, and client-side certificates. In order to ensure disk performance did not skew our results, we

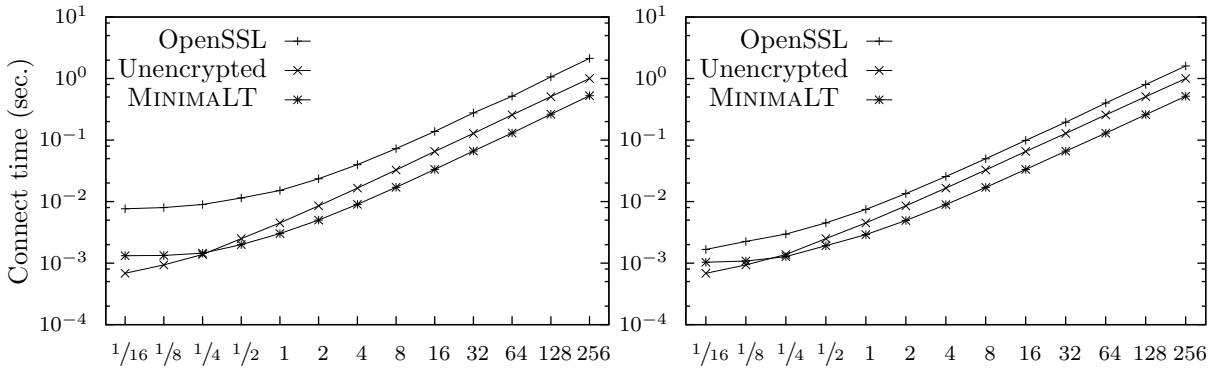
modified `s_server` to provide responses from memory instead of from the filesystem. We also wrote a cleartext benchmark which behaves similarly, but makes direct use of the POSIX socket API, avoiding the use of cryptography.

We benchmarked MinimaLT on Ethos. To produce results analogous to OpenSSL, simulating both (1) many abbreviated connection requests to one server and (2) many full connection requests to many servers, we tested both (1) the vanilla MinimaLT stack and (2) a MinimaLT stack we modified to artificially avoid tunnel reuse.

Figure 14a displays, in log scale, the rate of MinimaLT and OpenSSL when creating full, client-user-authenticated connections. For each connection, MinimaLT creates a new tunnel and authenticates the client user, and OpenSSL performs a full handshake; each requires public-key operations.

At native LAN latencies plus $1/16$ ms (LAN+$1/16$ ms), MinimaLT took 1.32ms to complete a full connection, request, and response, and OpenSSL took 7.63ms. MinimaLT continued to outperform OpenSSL as network latency increased. At LAN+256ms, MinimaLT took 526.31ms, while OpenSSL took 2.13s.

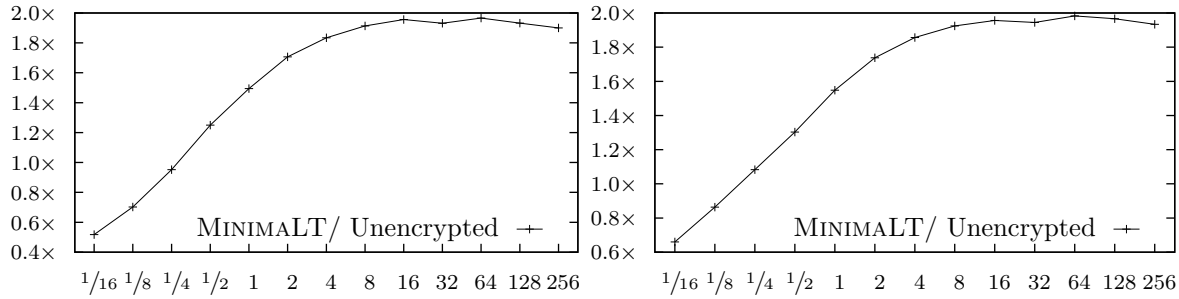
Figure 14b displays abbreviated connection speed. In this case, MinimaLT reuses an already established tunnel and OpenSSL takes advantage of its session ID to execute an abbreviated connection. Here both systems avoid computing a shared secret using DH, except in the case of the first connection. At LAN+$1/16$ ms, MinimaLT took 1.03ms to complete a connection, request, and response over an existing tunnel. Under the same conditions, OpenSSL took 1.67ms



One-way additional latency applied to network (ms), in addition to our native LAN latency of 1/13 ms

(a) Time spent creating a full connection

(b) Time spent creating an abbr. connection



One-way additional latency applied to network (ms), in addition to our native LAN latency of 1/13 ms

(c) Full MinimaLT improvement over TCP/IP

(d) Abbr. MinimaLT improvement over TCP/IP

Figure 14: Time spent creating a connection when using MinimaLT, cleartext TCP/IP, and TLS (less is better)

to complete an abbreviated connection, request, and response. At LAN+256ms, MinimaLT took 517.24ms, while OpenSSL took 1.60s.

In all measurements, MinimaLT connections incur less latency than OpenSSL. More surprisingly, MinimaLT creates connections faster than raw TCP/IP beginning before LAN+1/4 ms latency in the case of abbreviated connections and beginning before LAN+1/2 ms latency otherwise. We experience latencies of at least this magnitude on any packets which leave our

TABLE IV: MinimaLT connection rate with many clients

Tunnels per run	User Auth.	Connections per second	DH per conn.
One		18,453	0
One	✓	8,576	1
Many		7,827	1
Many	✓	4,967	2

laboratory room (i.e., are processed by our router). Figures 14c and 14d show the ratio between MinimaLT and raw TCP/IP performance. We attribute our results to MinimaLT's efficient tunnel/connection establishment (especially at high latencies) and to the speed of the NaCl library (especially at low latencies).

4.2.2.2 Tunnel establishment throughput with many clients

We next measured the CPU load on a MinimaLT server servicing many clients. To do this, we ran two client Ethos instances, each forking several processes that connected to the server repeatedly as new clients; here each virtual machine instance was running on a single computer. Because this experiment concerns CPU use and not latency, these clients do not write any application-layer data, they only connect. Using xenoprof and xentop, we determined that the Ethos server was crypto-bound (i.e., 63% of CPU use was in cryptography-related functions—primarily public key—and the server CPU load was nearly 100%). We measured the number of full connections per second achieved under this load, and varied the Ethos configuration from accepting fully anonymous users (no authenticators), to verifying a new user authenticator for each connection request. MinimaLT established 4,967–7,827 tunnels per second, as shown in Table IV (rows 3–4).

Given the minimal tunnel request size of 1,024 bytes, our hosts can (on a single core) service 61.15Mb/s of tunnel requests from anonymous users and at least 38.80Mb/s of tunnel requests from authenticated users. We note that this is the worst case for authenticated users. In general, we would cache valid user authenticators, thereby avoiding DH operations. Thus in practice we expect the authenticated user case to approach that of anonymous users.

4.2.2.3 Connection establishment throughput with many clients

We repeated the previous experiment, but this time repeatedly used a single tunnel between each client and the server. Table IV (rows 1–2) shows that our rates ranged from 8,576–18,453 per second, depending on the presence of user authenticators. The connection rate over a single tunnel is important for applications which require many connections and when using many applications to communicate with the same server. Our comments about cached authenticators in the preceding experiment applies here as well; we would expect in practice the rate of the authenticated case will approach the anonymous case.

4.2.2.4 A theoretical throughput limit

We used SUPERCOP [157] to measure the time it takes our hardware to compute a shared secret using DH, approximately 293,000 cycles or 14,000 operations per second. MinimaLT's tunnel establishment rate approaches 56% of this upper bound, with the remaining time including symmetric key cryptography, scheduling, and the network stack.

4.2.2.5 Single-connection data throughput

Finally, we measured the network throughput between two Ethos hosts. Table V describes our throughput results, observed when running programs that continuously transmitted data on

TABLE V: Comparison of MinimaLT throughput with other protocols (ignoring protocol overhead)

System	Bytes per second
Line speed	125,000,000
Unencrypted	117,817,528
MinimaLT	114,085,068
OpenSSL	95,605,458

a single connection for thirty seconds. MinimaLT on Ethos approaches the throughput achieved by cleartext networking on Linux and transmits payloads (not counting protocol overhead bytes) at 91% of line speed (Gb/s). Indeed, MinimaLT's cryptographic operations run at line speed; header size differences are the primary reason the cleartext benchmark outperforms MinimaLT.

4.2.3 Denial of service

DoS protections in MinimaLT are intended to maintain availability against much more severe attacks than are handled by current Internet protocols. Of particular concern are DoS attacks which consume memory or computational resources; the protocol cannot directly defend against network exhaustion attacks, although it avoids contributing to such attacks by preventing amplification.

We introduced anonymous and stranger-authorized services in §4.1.1.1. Anonymous services (i.e., permit create_0) must perform a DH computation to compute a shared secret and maintain tunnel data structures that consume just under 5KB each (this is configurable, most memory use is due to incoming and outgoing packet buffers). Stranger-enabled services (i.e, require $\mathsf{createAuth}_0$, but permit strangers) must additionally perform a public-key decryption to validate each new user authenticator encountered.

4.2.3.1 Before establishing a tunnel

In the case of anonymous services, a single attacker could employ a large number of ephemeral public keys to create many tunnels, with each tunnel requiring a DH computation and consuming memory on the server. Furthermore, the attacker's host might avoid creating a tunnel data structure or performing any cryptographic operations, thus making the attack affect the server disproportionately.

As discussed in §4.1.3.1, MinimaLT addresses these attacks using puzzles present in its tunnel headers. Servicing tunnel requests in excess of the limits discussed in §4.2.2 would cause a server to require puzzles. Ethos can generate and verify 386,935 puzzles per second on our test hardware. Since a puzzle interrogation and padded solution packet are 206 and 1,024 bytes, respectively, a single CPU core can handle puzzles at 394% of Gb/s line speed.

At tunnel establishment, MinimaLT may respond to packets from clients which spoof another host's IP address. This is always the case with the directory service, which initially must react to a request from an unknown party before transitioning to PFS-safe authentication. A MitM could spoof the source of packets, even while completing a puzzle interrogation. A weaker attacker could elicit a response to the first packet sent to a server. Given this, MinimaLT is designed to minimize amplification attacks. Amplification attacks cause replies from a legitimate service to be sent to a spoofed source address, and are compounded when protocol replies are larger than requests. A MinimaLT connection request causes a connection acknowledgment or puzzle interrogation; both responses are smaller than the request.

4.2.3.2 After establishing a tunnel

Given a tunnel, an attacker can easily forge a packet with garbage in place of ciphertext and send it to a service. This forces MinimaLT to decrypt the packet and verify its checksum, wasting processor time. However, MinimaLT's symmetric cryptography on established tunnels operates at line speed on commodity hardware (§4.2.2), so DoS would be equivalent to the attacker saturating a Gb/s link.

Low cost (small w) puzzles can be used to check that a client is still reachable. Puzzles can occur at other than connection establishment time, so they can require that a client perform work to keep a connection alive. We use control connection RPCs to pose and solve these puzzles to prevent an attacker from attempting RST-style mischief [140].

Stranger services are vulnerable to further CPU attacks—attackers could generate false user identities that would fail authentication, but only after the server performed a public-key decryption. A server will apply the puzzle RPCs when connection rates exceed the limits discussed in §4.2.2.

An attacker could also generate verifiable authenticators and connect to a stranger-authorized service many times as different stranger users. This would cause Ethos to generate sparse accounts for each stranger user. However, this is no different from any other creation of pseudo-anonymous accounts; it is up to the system to decide how to allocate account resources to strangers. Perhaps the rate is faster because of the lack of a Captcha, but unlike many contemporary pseudo-anonymous services, Ethos can prune stranger accounts as necessary; the stranger's long-term resources (e.g., files on disk) will remain isolated and become available if

the account is later regenerated because public keys remain temporally unique [95]. Of course, applications could impose additional requirements (e.g., a CAPTCHA) before allowing a stranger to consume persistent resources like disk space.

4.2.4 Ongoing performance tuning

Using a single CPU core, MINIMALT transmits encrypted data at nearly one Gb/s and performs thousands of authentications per second. Future work will focus on increasing the rate of the public key operations inherent to tunnel establishment through the use of multiple CPU cores. We expect a roughly N-fold improvement in cryptography from using N cores, and thus expect Gb/s-speed anonymous tunnel establishment using 16 cores and Gb/s-speed authenticated connection establishment using 25 cores. These represent upper bounds, where incoming traffic consists only of tunnel establishment packets. In reality, traffic will contain other requests, and Ethos caches valid authenticators to speed up authenticated connections.

CHAPTER 5

AUTHENTICATION

The abstractions used to provide OS authentication are fundamental to a system's security. Despite this, integrating authentication in a way which is simple, general, and robust against attack has proven elusive. Here we describe how we integrate the authentication facilities provided by MinimaLT into the semantics of Ethos' network-authentication-related system calls, as well as how Ethos provides for local authentication (because this is a prerequisite to network authentication).

Systems must authenticate both local and remote users, but subtle vulnerabilities can be a consequence of system design. In *authentication privilege escalation* [127], a process gains unintended privileges using authentication primitives. Here there are two possibilities:

(1) Processes p_0 and p_1 are owned by different users; p_0 crafts p_1's state to trick p_1 into doing something.

(2) A process acquires elevated privileges for authentication and is attacked before it drops them.

For an example of (1), p_0 might set an environment variable to specify the filesystem location from which dynamic libraries are loaded and then exec a setuid-program p_1 [158]. In fact, p_0 can craft p_1's state in many ways, including environment variables, file descriptors, and command-line inputs. A naïve programmer may be unaware of such attacks, and even sophisticated

programmers make mistakes. Libraries pose particular problems because of unavailable source code and revisions that affect previously installed programs. In (2), elevated privileges increase the impact of vulnerabilities.

Ethos introduces new, safer-to-use authentication mechanisms. Ethos' contributions to authentication are as follows:

- Ethos authenticates all network requests before any application code runs.
- Ethos is free from authentication privilege escalation.
- Ethos has new, simple-to-use authentication system calls.
- System administrators rely on system-level authorization and network authentication, and thus do not need to audit application code for these properties.

In addition, Ethos guarantees high confidentiality of authentication secrets, scales to the Internet using public-key cryptography, supports multiple identities per user, and supports anonymity.

5.1 Ethos authentication design

Ethos authentication is deeply integrated into the system's networking and process mechanisms: public key cryptography is integrated with networking, password-based local login is differentiated from cryptographic-based network authentication, cryptographic keys are managed by Ethos, and both local and network authentication use a common mechanism for creating credentialed processes. Ethos authenticates all local and remote users.

An Ethos host uses two different authentication services. A Local Authentication Service (LAS) contains the name, User ID (UID), and hashed password of each user who can physically log onto the system, and each host maintains its own LAS. A Remote Authentication Service (RAS) contains user names and their public keys, server names and their public keys (i.e., the directory service described in Chapter 4), and groups. Multiple RASs form a distributed Public Key Infrastructure (PKI). Unlike with Kerberos, the compromise of a RAS does not compromise individual users' private keys because a RAS does not have access to them.

An Ethos *host keystore* holds the host's own public and private keys as well as the public key (root of trust) for the RAS. For each local user, Ethos maintains a *user keystore* that holds his public and private keys [98]. Ethos' keystores drew inspiration from Plan 9's secstore, the difference being that keystores are part of the Ethos kernel. The Ethos keystores isolate secret keys as described in Chapter 3, and further protect user keys because Ethos will not allow their use (indirectly, through system calls) except for by their locally-authenticated owner.

All users are visible to Ethos. Ethos services do not maintain application-level user databases, as does MySQL or Apache with mod_authn_file. A user's application interaction in Ethos can be customized solely through authorization (Chapter 8).

5.1.1 Authentication system calls

The most interesting part of Ethos is its system calls, which target a higher level of abstraction than POSIX. The most relevant to authentication are the nine system calls in three categories shown in Table VI. As in POSIX, the child of an Ethos fork inherits the authentication credentials of its parent. However, Ethos authentication mechanisms are quite different

TABLE VI: Authentication-related system calls in Ethos

Process	
fork	create a child process
exec	change process executable
exit	terminate a process
Authentication	
authenticate	authenticate a local user
fdSend	send a file descriptor
fdReceive	receive a file descriptor
Networking	
advertise	offer a service
import	accept a connection to a service
ipc	connect to a service

from those of POSIX: there is no setuid system call and the user associated with a process never changes (§5.1.2). In addition, exec cannot change the authentication credentials of the process—Ethos does not have a setuid bit. Instead, Ethos provides fdSend/fdReceive (§5.1.2).

Ethos networking and local IPC both use the same system calls: advertise, ipc, and import (§5.1.4). Network communication is encrypted and protected against tampering by cryptographic checksums (Chapter 4). IPC communication remains as cleartext, but Ethos protects it against observation and tampering by memory management. In either case, all communication is subject to authorization on each host participating in the communication.

For local authentication, where the user is physically present at the computer, Ethos provides an authenticate system call to establish a user's identity by password (§5.1.3). Passwords are a relatively weak mechanism [159]. However, local authentication requires physical proximity

to the computer, eliminating access by the vast number of (remote) attackers. If stronger local authentication is required, multi-factor authentication using a physical token or biometric can be used.

Remote authentication is inherently different than local authentication. Here, direct use of smart cards and biometrics are inappropriate, and passwords are unreliable, due to spoofing, key logging, and network timing attacks [63; 64]. Furthermore, large botnets—containing upwards of a million nodes—allow for a staggering number of password attempts, and have been very effective [1]. Thus Ethos' ipc/import guarantee that all network connections are authenticated using public-key cryptography (§5.1.4).

5.1.2 Virtual processes

Ethos' authentication mechanisms are centered around its *virtual processes*, per-user processes created on demand. To invoke a virtual process, an application sends a tuple of file descriptors (typically, for a network connection) to it and specifies its user. The API to send descriptors is:

```
fdSend(fd[], user, program)
```

where fd[] is a tuple of file descriptors, user is the user that will own the virtual process, and program is the file containing the virtual process' executable code. fdSend's tuple of file descriptors simplifies its failure semantics—sending a sequence of file descriptors will either completely succeed or completely fail. On Ethos, the process which calls fdSend serves as a distributor, as it distributes file descriptors to the appropriate user's virtual process.

After a virtual process has been invoked by fdSend, the virtual process can receive the sent file descriptor tuple using fdReceive:

```
fds ←fdReceive()
```

Naturally, virtual processes require careful authorization. Ethos virtual processes are used in one of two ways: Typically, when sending a network file descriptor, a distributor sets user to the remote user associated with the connection. We call this a *remote-user virtual process*. Alternatively, the distributor can set user to some user other than the remote user. We call this an *arbitrary-user virtual process*. The latter allows for local authentication and mail-like messaging programs (where the owner of the virtual process is the message recipient, rather than the message sender/remote user). We will describe the authorization support for both remote-user virtual processes and arbitrary-user virtual processes in Chapter 8 and present examples of its use in Chapter 9.

Ethos is free from authentication privilege escalation. Because Ethos processes never change their owner, their privileges do not change as a result of authentication. A virtual process is not created using fork, so it has no predecessor process. In that sense, it is like init on UNIX systems. Thus the environment of a virtual process is defined by Ethos and is unaffected by any process; moreover, only the appropriate distributor may invoke it with fdSend. Finally, the permission for authentication is narrowly scoped: it does not allow extraneous actions.

5.1.3 Local authentication

Ethos permits authentication by password (or physical token/biometric) only when the user can physically interact with the computer—never over the network. For obvious reasons, we call this *local authentication*.

The API to authenticate a user such as *Alice* locally is:

```
authenticate()
```

The authenticate system call establishes a trusted path [34; 160] and prompts *Alice* for her password. *Alice* must input a password locally (not over the network), and the password is never seen by the application. Ethos notes *Alice* as the owner of the calling process and then verifies her password; based on this check, authenticate returns true (authenticated) or false (rejected). Ethos rate limits authentication and logs its use, providing resistance to excessive local password attempts at the system level.

Authenticate does not require any privilege. It is safe for any application to invoke, as it does not disclose passwords to the application or change the process' OS state (as does, e.g., POSIX's setuid).

Figure 15 shows the key code in Ethos' login program, made up of two programs totaling 68 lines of Go. *Alice* first provides her name to the login distributor (Line 2). The distributor passes this name to fdSend, invoking the virtual process loginVP and providing the terminal's file descriptors (Line 3). (stdinFd, stdoutFd, and stderrFd correspond to POSIX's 0/stdin, 1/stdout, and 2/stderr.) After receiving the file descriptors with fdReceive, loginVP calls the authenticate

```
1 do forever
2      user ← read (stdinFd);
3      fdSend ([stdinFd, stdoutFd, stderrFd], user, "loginVP")
```

(a) Distributor distributes terminal fds to virtual processes

```
4 stdinFd  ← fdReceive ()
5 stdoutFd ← fdReceive ()
6 stderrFd ← fdReceive ()
7 if authenticate()
8      // User authenticated; exec shell.
```

(b) loginVP checks password and then does user startup

Figure 15: Ethos local authentication in pseudo code

system call (Line 7). Of course, loginVP is already running as Alice when it makes this call; the call to authenticate simply means that login will not proceed without a valid password.

5.1.4 Network authentication

Ethos cryptographically authenticates all network connections, so applications cannot fail to authenticate, authenticate a network connection using a password, or incorrectly authenticate. Ethos' use of public-key cryptography means that authentication does not require the sharing of secrets (as with Kerberos), users can create their own key pairs, and each public key is guaranteed unique. Because of the last property, public keys are UUIDs that can serve as UIDs [128; 95; 144] as in Taos, even if the user's real-world identity is not known. This enables uniform support of stranger, anonymous, and identified principals. Ethos is different than Taos because Ethos authenticates users without requiring an application to call a procedure like GetPrin. For these reasons, Ethos reduces the opportunities for application-based subversion of network authentication.

For all IPC—including networking—Ethos uses a single mechanism, whereas POSIX provides sockets, pipes, message queues, shared memory, signals, and so on. A single mechanism simplifies Ethos' design, application programming, and access controls. IPC is available through the advertise, import, and ipc system calls, which we now describe.

```
serviceFd ←advertise(serviceName)
```

To register the intent to provide a service, an Ethos application calls advertise. The argument to advertise is the service name.

```
netFd, user ←import(serviceFd)
```

The import system call takes as an argument a service file descriptor that was obtained from a previous call to advertise. Ethos cryptographically authenticates the remote user associated with an incoming network request, and subjects the user to its authorization policy (Chapter 8). Thus import only returns for network requests from authorized (and authenticated) users, and it can return to the calling program both a network file descriptor and the remote user. In the case of known users, the returned string is the user's name; for strangers and anonymous users, import returns a string representation of the user's public key. Thus import resembles Taos' GetPrin, but Ethos performs mandatory authorization, so an analogue to Taos' Check is not necessary.

```
netFd ←ipc(serviceName, host)
```

An ipc call attempts to make an encrypted connection to a service. This system call takes two parameters: serviceName—the service name to connect to—and host—an optional remote host name. Ethos resolves host to an IP address and public key as a part of servicing the system

call (Chapter 4), or it is nil in the case of a local connection. A variation of ipc, ipcWrite, allows an application to send data on the first packet, as enabled by MinimaLT.

```
data ←read(fd)
data ←peek(fd)
write(fd, data)
```

Read reads an object from a file descriptor. Peek behaves like read but does not remove the object from the file descriptor's stream. (The necessity of peek is related to arbitrary-user virtual processes as we will describe in Chapter 9.) Write adds an object to a stream.

Figure 16 depicts an Ethos client initiating a connection to a distributor. The distributor accepts connections and distributes them, via fdSend, to the appropriately user-credentialed virtual process.

5.1.5 Principals, strangers, and anonymity

An Ethos user can assume different principals when making network connections on a per-host-and-service basis. This supports different identities for work, entertainment, and socialization. A principal can be a stranger (i.e., not present in a RAS), and anonymous principals result from creating a new identity on each connection to a service. Whether a service allows strangers or anonymous users is a matter of authorization (Chapter 8), independent of any ap-

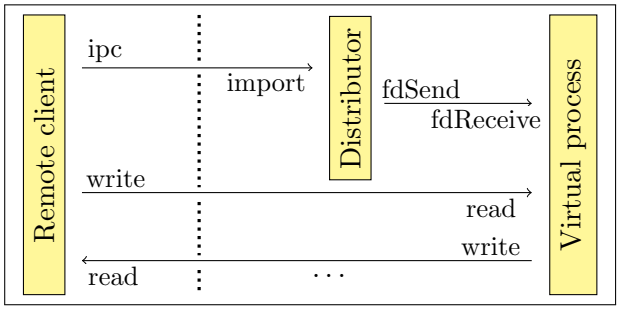


Figure 16: Interaction of processes in an Ethos client-server application

plication code. Ethos can isolate the persistent resources of strangers as with identified users, and assuming strangers protect their private key, no one else can assume their identity during authentication.

Strong anonymity is difficult to achieve, and depends on its use. Anonymous services need, in addition to anonymous users, further protections against de-anonymizing, such as with Dissent [161] or Tor [35]. This level of anonymity is not included in Ethos at the OS level due to its latency, but Ethos provides a strong foundation on which to build such services.

5.1.6 Implementation

Ethos maintains a hash table of user records. It initially populates this table with the users in its LAS. Ethos adds on demand other users to the table when it imports their network connections. In this case, Ethos extracts the user's UID (public key) from the authenticator present in MinimaLT and extracts his name from the RAS, if it is known. If the user is not known to the RAS, then Ethos sets his name to the string form of his public key (this is a stranger or anonymous user). User records also contain the user's running virtual process list.

When a process calls fdSend, Ethos first checks whether the running process is authorized to run the particular virtual process specified by the system call. Ethos then checks the user record to determine if the target virtual process is already running. If not, Ethos creates it by allocating the process' page tables, switching to the process' address space, setting the process' UID to the appropriate public key, loading the program image into memory, nullifying all file descriptor table entries, and allocating a heap and stack. Finally, Ethos adds the fdSend file

descriptor tuple argument to the virtual process' incoming fdReceive queue. Ethos will then schedule the virtual process which can obtain the file descriptors by calling fdReceive.

5.1.7 In brief

Ethos provides a small number of authentication mechanisms, each designed to be strong and easy-to-use. The authenticate system call allows for local authentication without sharing authentication secrets with applications. An Ethos process and its descendants share a single, immutable user, but processes can invoke per-user virtual processes through the use of the fdSend and fdReceive system calls. Virtual processes are executed by the kernel, as with init, and thus their environment cannot be influenced by another program.

Authentication is also built into the ipc and import system calls, so network authentication is both implicit and cryptographically strong. Finally, the design of Ethos includes the notion of stranger and anonymous users. This means that despite implicit network authentication, Ethos hosts can interact with the Internet at large, even though some users will not be known to Ethos before they connect.

5.2 Evaluation

In Chapter 9, we will present a series of case studies which demonstrate the effect of Ethos' security properties on applications. Here we focus on Ethos' authentication-related security properties and compare them to other systems. We also describe the performance of Ethos' authentication primitives, which we evaluated using the same computers described in Chapter 4. We ran all of our POSIX comparisons with nscd, to minimize the latency of authentication database queries.

TABLE VII: Security features of Ethos authentication; BP denotes a best practice shared with another OS, N denotes a novel feature, and the primary security property (as defined in Chapter 1) that benefits from each feature is noted in parenthesis

Ethos		**Comparison**
Authentication privilege escalation avoidance		
A process p, running as user u_1, cannot influence the environment of an invoked process p', running as user u_2. (P1)	N	On POSIX, it is possible to set environment variables, reset the standard file descriptors, etc. before executing a setuid-bit program. Other OSs avoid the setuid-bit, but have similarly fragile mechanisms: SELinux provides a setexeccon system call and processes can also transition to a new domain upon execution based on their executable's label. On Plan 9, one program can execute another, and the invoked program can change its privileges through the use of factotum and the capuse device.
FdSend does not require superuser privileges. (P6)	BP	setuid requires root privileges, and a program that runs as root until it calls setuid can be attacked before transitioning to the new user. Ethos resembles Plan 9, where transitioning to a new user is a capability.
Owners of Ethos processes never change. (P1)	BP	POSIX process ownership is complex, as defined by setuid, seteuid, and setreuid. Ethos resembles Singularity, which also has immutable process owners.
Local authentication		
The authenticate system call ensures applications do not have access to secrets. (P1)	N	On POSIX, authenticating applications have access to user secrets. Plan 9 provides auth_userpasswd to validate a password, but the application must read the password itself and provide it to auth_userpasswd. Singularity likewise requires sharing secrets with programs that perform local authentication.

Network authentication		
Ethos performs mandatory network authentication as a part of the import/ipc system calls. (P1)	N	On POSIX, Plan 9, and Taos, network authentication must be explicitly invoked by applications.
Ethos forbids Password-based network authentication. Ethos uses only cryptographic network authentication. (P1)	BP	POSIX and Plan 9 allow both weak and strong authentication techniques. Ethos resembles Taos because both provide a single, strong network authentication technique.
Isolation of secrets		
The Ethos kernel naturally isolates secrets required by both local and network authentication. (P4)	N	On Plan 9, factotum must be protected from examination (e.g., by a debugger or by inspecting the /proc filesystem) and never swapped to disk. Furthermore, Plan 9 applications have access to secrets when performing local authentication. On Singularity, applications perform their own authentication.
Authentication database		
Ethos provides for a strong trust model, where each user can choose their TTPs, and mutual authentication can take place without symmetric TTPs. (P1)	BP	Kerberos and hierarchical X.509 imply symmetric trust amongst all parties. Ethos provides a trust model that more resembles SDSI or PGP.
A RAS compromise does not directly compromise user secrets. (P1)	BP	The compromise of a Kerberos TGS compromises the secrets of all users; public-key-based authentication is more robust.
A RAS cannot masquerade as another user. (P1)	BP	The Kerberos system has the ability to masquerade as any user; public-key-based authentication is more robust.
User identity		
Ethos provides a coherent user identity model that spans the entire Internet. (P1)	BP	POSIX and Plan 9 must map 32-bit integer/string user identifiers to Internet identities in an ad-hoc manner. Ethos resembles Taos, which also uses public keys as user identifiers.

Ethos is designed to isolate the resources (e.g., files) of individual strangers, even though their real-world identity is not known. This requires no administrator intervention, other than configuring a service to permit strangers.	N	Existing systems do not integrate system-wide support for strangers.

5.2.1 Security

Table VII summarizes the authentication-related security features provided by Ethos. Ethos addresses authentication privilege escalation, provides a simple-to-use authenticate system call for local authentication, makes strong network authentication a mandatory protection provided by import/ipc, carefully isolates user secrets, and provides a coherent user-identity model that spans the entire Internet. Combined, these features minimize the possibility of application-based subversion of authentication by removing the responsibility of performing authentication from most applications. Indeed, no Ethos application receives any network message that has not been authenticated by Ethos. Furthermore, Ethos' RAS provides for a stronger trust model than Kerberos or hierarchical X.509.

5.2.2 Performance

We measured the performance of setuid on POSIX with two programs, linuxSetuid and linuxSetuidExec. To these we compared ethosFdSend, an Ethos program. We padded each program to be approximately 800,000 bytes. We ran each benchmark several times, varying the percentage of repeat users. On Ethos this means that at one extreme each call to fdSend invokes

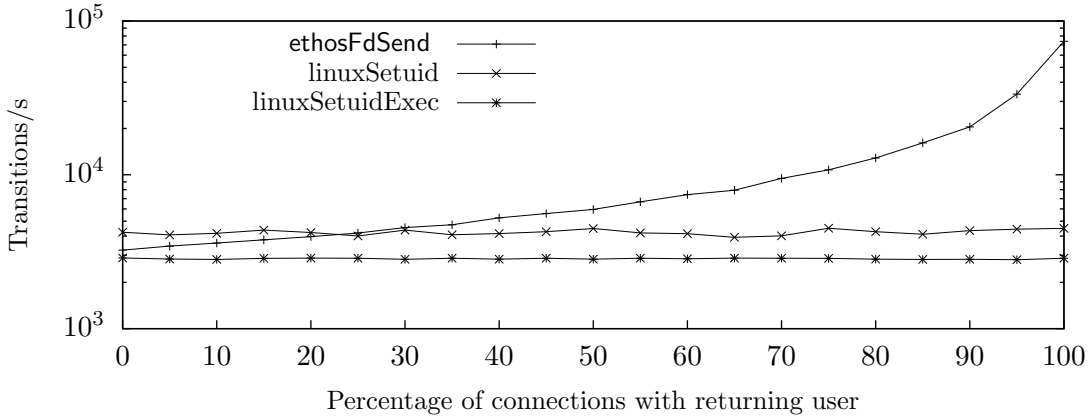


Figure 17: User transition rate: fdSend vs. setuid

a virtual process for a new user, whereas at the other extreme the benchmark repeatedly sends a file descriptor to a single user's virtual process.

LinuxSetuid repeatedly forks and in the child process transitions with setuid to a user/users known at compile time (but resolved at run time to a UID using getpwnam) before exiting. LinuxSetuidExec extends linuxSetuid by calling exec after setuid. This better isolates the post-authentication logic and allows controls—such as SELinux—to restrict the server separately from the distributor. This trades performance for robustness.

EthosFdSend is a program that measures the performance of fdSend, including both creating the virtual process and sending it a file descriptor. ethosFdSend loops for a set period of time, calling fdSend with some user on each loop iteration. Each invoked virtual process runs a loop that continuously tries to fdReceive file descriptors.

Figure 17 displays the results of comparing linuxSetuid, linuxSetuidExec, and ethosFdSend. This figure plots the performance in user transitions per second across a range of repeat user percentages. From left to right, the figure depicts performance from the case of 2,000 iterations

with no returning users (0%) to the case of a single user serviced 2,000 times (100%). When ethosFdSend encounters a new user, Ethos creates a new virtual process; loading this program is the primary source of latency. On the other hand, if a user returns, then the user's virtual process already exists, lowering latency.

In the worst case, where each user encountered is new, ethosFdSend was able to create 3,243 virtual processes per second. When all users are repeats (i.e., high user locality), its throughput was 73,803 user transitions per second. EthosFdSend always outperforms linuxSetuidExec, and this performance advantage ranges from 113% to 2,573%. At approximately 25% returning users, ethosFdSend outperforms linuxSetuid. The relative performance of ethosFdSend to linuxSetuid ranges from 77% to 1,645%.

CHAPTER 6

TYPES

Type-safe programming languages leave many seams with respect to security: distributed systems are often written in multiple languages, and their applications must be integrated. Even for distributed applications written in a single language, it is beneficial to ensure a consistent use of types. Programming language type systems primarily provide internal program consistency. In contrast, OS-based types can constrain input and output to conform to program expectations. Thus we are interested in constructing an OS which provides aid at the seams between applications, which themselves are written in type-safe languages.

The most compelling reason for adding types to an OS is for security. Existing distributed systems often act insecurely due to ill-structured input and output. Many applications dedicate a significant amount of code to parsers (or in some cases decoders) that translate untyped data into a language's native form. Attackers often can craft input to exploit vulnerabilities in this code. An attacker might also fool a program into generating data that it should not.

Traditional OSs are untyped and have untrapped errors with respect to applications. The result is that applications written on top of such OSs can fail silently and in unpredictable ways. Some OSs use type safety for internal consistency, to eliminate errors within the OS, or at the OS interface. For example, dynamic checkers have been used to prevent buffer overflows within kernels. Static type checkers have been used to regulate either internal RPCs or code extensions.

Implementing an OS and its applications in a type-safe language gives rise to new techniques for providing protections. Java's type system provides isolation within the JX OS [162]. JX OS does not require a Memory Management Unit (MMU) and instead implements software-based stack overflow and NULL reference detection. Fabric is also based on Java. Fabric labels objects with their security requirements and provides distributed-system-wide, language-based access and flow controls [163]. Singularity is written in Sing♯, provides SIPs, and uses linear types to prevent shared-memory-related race conditions [164]. LISP and Haskell have also been used to construct an OS kernel [165; 166].

SPIN allows applications to specialize the OS kernel using extensions [167; 168]. Type safety allows SPIN to dynamically link extensions into the kernel while isolating other kernel data structures. SPIN extensions are written in Modula-3. Exokernels likewise maximally embody the end-to-end argument [29]. This has been shown to provide performance benefits, for example by allowing applications to interact more directly with network interfaces [169]. Unikernels are single-purpose microkernels written in the type-safe language OCaml and designed to run within a hypervisor [170].

Ethos takes a different approach to type safety. We call Ethos' type system *Etypes*. Processes send data to other processes—either directly through IPC or indirectly through the filesystem—in the form of *Ethos objects*, and objects are typed. Each object has at least two representations, one in a running program and another external representation. Etypes converts between external serialized objects and internal program memory and Ethos subjects all process input and output to a type checker.

Ethos' type checker (§6.2.3) assures a property we call object integrity. We define *object integrity* as:

(1) objects are read or written in whole, thus preventing short or long reads/writes [171],

(2) objects (external or memory) must be consistent with their type, and

(3) an object which is written by one program and read by another must produce an *equivalent object* (see below).

A system call that writes/reads object o will either succeed if o is well-typed or return an error without application side effects. Managing side effects in this way simplifies application development, and is a strategy shared with other systems [172].

Our notion of object includes not only the object directly referenced (o_0), but also those objects which are reachable from o_0 via pointers ($o_1, o_2, \ldots o_n$). Given this numbering of objects, the reconstituted object must consist of n objects $o'_0, o'_1, \ldots, o'_n$, such that the type of o_i is equal to o'_i, and the value of o_i is equivalent to o'_i. Two objects o_i and o'_i are *equivalent* if (1) each pointer field in o_i that points to o_j has a counterpart field in o'_i that points o'_j and (2) each non-pointer field in o_i is equal to its counterpart in o'_i.

Ethos is the first OS to subject its system calls to a language-agnostic system-wide type checker. Programmers define types in advance of use, and Ethos restricts (in conjunction with authorization) what types programs can read or write. This limits the attacker's ability to introduce ill-typed data and reduces surprises—and therefore vulnerabilities—during application reads. That Etypes is language-agnostic is significant because experience has shown that

single-language or single-language-runtime systems are insufficient [84]. Etypes' contributions include:

C1 UUIDs for types without naming authorities (§6.2.4). Types are independently defined, guaranteed not to conflict with types created elsewhere, and usable anywhere. This is necessary for loosely coupled [173] distributed systems.

C2 Defining the types of files and IPC enables OS-based type checking of all process input and output (§6.2.3).

C3 Tightly integrated programming language/system support (§6.2.2, §6.2.3). This integration reduces application code, eliminates type errors, and impacts the design of scripting languages.

An application might still violate more sophisticated restrictions on object use. For example, an integer might represent time, which should increase monotonically. Or, a type might contain pointers but prohibit cycles. In Etypes, these considerations are left to applications—there is an inherent trade off between generality and safety in a system-wide type system. We call these higher-level semantics *business rules*. Although business rules are the responsibility of applications, they can be described with a type's Etypes annotations (§6.1.2). Thus an application developer has documentation of the business rules expected to be enforced by his application.

6.1 Etypes

With Etypes, programmers specify types and RPC interfaces using Etypes Notation (ETN) which, like XDR, is a data description language (§6.1.1). Etypes' serialized wire format is called Etypes Encoding (ETE). Etypes has three fundamental operations: *encode*, which takes a programming language type and serializes it to an ETE; *decode*, which takes an ETE and sets an appropriately typed programming language variable to its value; and *check*, which Ethos uses to implement its type checker. Etypes addresses both the syntax and semantics of types (§6.1.2).

6.1.1 Etypes Notation

ETN describes types using a syntax based on Go. Table VIII lists the ETN types, a syntax example for each, and their corresponding ETE. Primitive types include integers (both signed and unsigned), floats, and booleans. Composite types include pointers, arrays, tuples, strings, dictionaries, structures, discriminated unions, `any`, and RPC interfaces. RPCs, unions, and `any` types, warrant further description.

An Etypes *RPC interface* specifies a collection of RPC functions. Etypes RPCs are built from stub and skeleton routines, similar to ONC RPC or CORBA. Although ETN specifies RPC functions in a traditional way, their Etypes implementations are in fact one-way—they do not have direct return values. Instead, a callee returns a value by invoking a reply RPC; this supports both asynchronous and synchronous communication. One-way RPCs are a very simple, yet entirely general, mechanism.

TABLE VIII: Primitive, vector, composite, and RPC type ETN and ETE; UUIDs are encoded as arrays of bytes; lengths are encoded as uint32; T is an arbitrary type; ‖ is concatenation.

Type	ETN	ETE
Integer	b byte i int$X$ u uint$X$	little-endian signed or unsigned X-bit integers, where X is 8, 16, 32, or 64
Boolean	b bool	unsigned 8-bit integer
Floats	f float32 f float64	little-endian IEEE-754
Pointer	p ∗T	enc. method ‖ value
Array	a [n]T	values
Tuple	t []T	length ‖ values
String	s string	length ‖ UTF-8 values
Dictionary	d [T]S	length ‖ key/value pairs
Structure	N struct {...}	field values
Union	M union {...}	uint64 union tag ‖ value
RPC	F(T$_0$,T$_1$,...,T$_n$)	uint64 func. ID ‖ args.
Any	a Any	type's UUID ‖ value
Annotation	['text'] [see 'filename']	n/a: contributes to UUID

ETE is implicitly typed, with two exceptions: unions and the any type. A union may be instantiated to one of a specified set of types and an any may be instantiated to one of the set of all types. When a program encodes an object as either, Etypes includes the object's actual type identifier (§6.2.4) in the encoding.

6.1.2 Syntax and semantics

ETN's *annotations* informally describe semantics beyond types. Annotations

(1) contribute to a type's UUID, binding syntax and semantics together;

```
1  Certificate struct {
2      size          uint32
3      version       uint32
4      serialNum     uint32
5      validFrom     [16]byte
6      validTo       [16]byte
7      typeHash      HashValue
8      signaturePubKey  PublicSignatureKey
9      revocationServer [256]byte
10     signature     [128]byte
11     ['ABA transit number in MICR form']
12     bankId        uint32
13     ['From account; bank's num. std.']
14     fromAccount [] byte
15     ['To account; bank's num. std.']
16     toAccount     [] byte
17     ['Transfer amount in US dollars']
18     amount        uint64
19 }
```

Figure 18: An ETN structure representing a bank transfer certificate

(2) differentiate structurally identical types; and

(3) enable integrity requirements beyond typing to be expressed (i.e., business rules).

Application programmers, administrators, and users refer to a type's annotations to determine an object's meaning, and thus annotations minimize the semantic-level difference [103] as understood by Ethos users.

Consider the certificate type described in Figure 18, which is a digitally signed bank transfer order. It contains eight certificate header fields and four certificate-specific fields: a bank ID, two account numbers, and the transfer amount. Field names, i.e., bankId, contribute to type UUIDs but are insufficient for detailed descriptions. Annotations narrow the semantic-level difference, for example by describing the bankId field as an American Bankers Association

transit number (at line 10). More complex annotations can be in an external file referenced by an ETN specification.

6.2 Design

Targeting different programming languages presents both a challenge and opportunity. In §6.2.1, we describe how Etypes remains programming-language-agnostic through the use of type graphs, and we explain why multiple programming languages are needed to meet the various requirements of Ethos in §6.2.2. We discuss issues related to the deep integration of Etypes into Ethos in §6.2.3.

Etypes identifies its types using something we call a *type hash*—a UUID based on a cryptographic hash. The type hash is a fully distributed mechanism, naming each type in a unique but predictable manner, yet it does not require *any* naming authorities. Type hashes also support the requirement of versioning. We describe the algorithm to compute type hashes in §6.2.4.

6.2.1 Type graphs

Given an ETN specification, a utility named `et2g` generates a programming-language-independent type graph and its type hashes. A *type graph* is a directed graph containing type descriptions as nodes and type references as edges. A type graph identifies types by universally unique hashes rather than the local names used in its ETN specification. Type graphs are self-contained: for all types T in graph G, any type which T references is also in G. For example, if a struct appears in a type graph, so do the types present in the struct's fields.

Ethos forbids creating an OS object of type T unless T is present in the system's type graph (§6.2.3). Ethos uses the type graph to check types, and user-space utilities (such as `eg2source`

TABLE IX: The contents of a type graph node; one node exists for every type specified

Field	Description
Hash	Type hash, the UUID for the type
Name	Mnemonic for the type
Kind	Integer representing the type's kind (e.g., uint32, struct) from a fixed enumeration
Annotation	Annotation for a type, struct field, or func.
Size	Size, if the class is a fixed-length array
Elems	Tuple of type hashes representing: the type of a typedef, fields in a struct, type of elements in a vector, parameters/return values for an RPC function, RPC functions in an interface, or target of a pointer
Fields	Tuple of strings naming: the fields in a struct, parameters/return values for a function, or RPC interface functions

as described below) also make use of the same graph. Type graphs themselves are stored as ETE. Table IX shows the contents of a graph node and Appendix B describes type graphs in more detail.

6.2.2 Programming language integration

Our type system is programming-language-agnostic; nevertheless it has profound impacts on the programming languages Ethos supports. In particular, we discuss its impact in terms of three different types of programming languages: unsafe, statically-checked programming languages such as C; type-safe, statically-checked programming languages such as Go; and type-safe, dynamically-checked programming languages.

For performance reasons, Etypes minimally relies on introspection. Instead, it uses eg2source to generate code targeted to a specific programming language; given a type graph and output

language, the tool generates code used to encode and decode types. For RPCs, eg2source creates stubs and skeletons, similar to ONC RPC or CORBA.

6.2.2.1 Unsafe programming languages

Encode and decode require compile-time type definitions, due to C's static typing. On the other hand, types can be added to a running system, so the kernel's use of check requires that check be table-driven and thus able to verify unanticipated types. (Currently, encode and decode are also table-driven, but unlike with check, this is not a requirement.) eg2source generates tables which describe types, and a library called libetn walks these tables to encode, decode, or check the types. Since libetn depends only on external malloc- and free-like functions, it is easily integrated into both the OS kernel and programming language runtimes. Etypes simplifies kernel code by subsuming tedious manual encode, decode, and verification routines (Appendix E).

C is not type-safe. Using Etypes in C can have untrapped errors, such as a mismatch between an ETN type and C variable. However, since the use of C on Ethos is limited to system software, its use can be subject to more rigorous code inspection.

6.2.2.2 Type-safe, statically-checked programming languages

Go is type-safe and statically-checked; such languages are where ETN is aimed. Go is the primary application-development language on Ethos. Each Go program contains only a fixed number of compile-time-defined types. Our Go Etypes implementation directly generates per-type encode/decode routines using eg2source, unlike C's libetn- and table-based implementation. The code for using an arbitrary type T is shown in Figure 19. The code in Figure 19a creates

```
1  e,d ← myTypes.Ipc(hostname, serviceName)
2  e.WriteT(&t)
```

(a) Create a connection and transmit a typed value t; the programmer uses eg2source to generate both myTypes.Ipc and e.WriteT from ETN definitions

```
3  e,d,u ← myTypes.Import(serviceName)
4  t ← d.ReadT()
```

(b) Accept a connection and receive a typed value t; the programmer uses eg2source to generate both myTypes.Import and d.WriteT from ETN definitions

Figure 19: Go code to create/accept an IPC and read/write a value on Ethos; T is an arbitrary type

an IPC channel and sends the value t of type T on it. The code in Figure 19b accepts an IPC channel and receives an object of type T from it. The procedures Ipc, WriteT, Import, and ReadT are generated by eg2source; thus the system calls necessary to implement these operations (Chapter 5) are hidden behind typed APIs. In keeping with Ethos' goal of minimizing application complexity, Etypes calls requires no more application code than untyped I/O calls in other OSs. However, Ethos calls do more, and this reduces the total amount of application code.

6.2.2.3 Type-safe, dynamically-checked programming languages

Statically-checked languages are preferred for traditional applications, because they provide higher integrity. However, utilities often benefit from dynamic types. Consider traversing a filesystem recursively and displaying the contents of files; here types might not be known at compile time. We are building eL, a dynamically-checked language that will integrate with Etypes.

TABLE X: Type-related system calls in Ethos

Miscellaneous	
createDirectory	create a directory bearing the given label and type
I/O	
writeVar	write an object to a file
readVar	read an object from a file
write	write an object to a streaming file descriptor
read	read the next object from a streaming file descriptor

One of the great successes of UNIX are the text-based utilities used to manipulate system state. Since UNIX was designed, these utilities have been diminished as types have become richer. Generic utilities, which processes types dynamically, are essential to making Ethos accessible to system administrators and others. Etypes will enable UNIX-like utilities that manipulate richer data.

6.2.3 Operating system integration

Here we discuss the integration of Etypes with Ethos. In particular, we describe how Ethos associates a type with every streaming file descriptor or file and how Ethos uses such types to regulate system calls. Table X lists the type-related system calls we describe here.

6.2.3.1 Type checking overview

Ethos verifies all network reads and writes as well as file or local-IPC writes using check. Filesystem encryption ensures that file writes must go through the Ethos kernel. Ethos traps ill-formed objects when writing rather than reading to aid correctness and problem diagnosis. Such type checking, like authorization, prevents mismatches between processes.

6.2.3.2 Associating types with objects

Every object in an Ethos filesystem directory must be of the same type, and every directory has associated with it a type hash that determines this type. Since Ethos IPC services are named by filesystem paths (we describe this in more detail in Chapter 8), directories determine the types of both files and IPC connections. Thus a program can only read/write object types corresponding to directories it is allowed to access. While this increases the number of directories, it has the advantage that Ethos can enforce the type-safety of file creation transparently—a write does not need to specify a type. We provide an example Ethos program that accesses a file in §6.2.5.

In Ethos, types must be specified only when creating a directory, a relatively infrequent operation compared to file operations. Applications create directories with the createDirectory system call, which accepts as parameters the parent directory dirFd, directory name, authorization label, and typeHash:

```
createDirectory(dirFd, name, label, typeHash)
```

We expect directory creation to be primarily handled by administrative tools. If the directories are set up outside of a particular application, then the application and type policy are completely independent. When a directory naturally contains various types, applying the any or union types allow it to be used in the style of traditional directories.

6.2.3.3 Files

In Ethos, the contents of a file can be any Etypes object, from primitives (e.g., a 32-bit integer) to complex entities made up of multiple objects, (e.g., a tree).

Ethos provides a writeVar system call to write a file in its entirety—an Ethos file is not a streaming object.

```
writeVar(dirFd, fileName, contents)
```

The inverse of writeVar is readVar.

```
readVar(dirFd, fileName)
```

Rather than directly use the above system calls, application programmers use encode/decode interfaces (§6.2.5).

6.2.3.4 Seek

It may seem odd not to support seeking within files; after all, video files can span many gigabytes. But this is necessary for simple failure semantics—an object is either of the correct type and the readVar/writeVar system call succeeds or it is not and the operation fails with no application side effects.

When designing Ethos, we had originally sought to provide seek. But its semantics became complex when considering typed objects because

(1) objects have variable size and hence computing an offset to a well-formed sub-object is not straightforward,

(2) the encoding is not normally visible to applications which only see decoded values, and

(3) errors would not be detectable until the whole file was read, complicating error recovery.

(3) is particularly troublesome, as an application would likely introduce side effects into the system as it reads well-formed offsets, only to encounter an ill-formed offset later. At this point, error recovery becomes (unnecessarily) difficult.

Ethos applications store very large entities using directory streaming rather than very large files, as we describe next. Because individual files must be read in their entirety, Ethos places an upper bound on file sizes to conserve memory.

6.2.3.5 Streaming IPC and directories

In Ethos, IPC and directories are streaming entities, and Ethos enforces the type of each write. The write system call sends data out on a streaming descriptor:

```
write(descriptor, contents)
```

Likewise, the read system call reads from a streaming descriptor. Again, the data must conform.

```
read(descriptor)
```

We provide an example of Ethos IPC in §6.2.5. Each read/write on an IPC channel reads/writes the next object. Objects are read from one side of an IPC in the same order they were written to the other.

Streaming directories present a particular challenge, because their files must be named to preserve their order. The write system call, when applied to a directory, creates a file named with the current time; the read system call, when applied to a directory, reads the next file in filename order. Thus Ethos streams over a directory of files (e.g., video frames), whereas each file is non-streaming (i.e., read in its entirety).

6.2.3.6 Networking

Etypes takes care of many networking chores at the OS-level including endianness, alignment, parsing, and data value encoding, so that networking just works. Because network data comes from a foreign system, Ethos applies check to network reads as well as writes. Higher-

level interfaces also allow Etypes optimizations independent of application code. No Ethos application can receive ill-typed data from the network, because first the data must satisfy Ethos' type checker.

6.2.3.7 Type graph

Ethos stores the system type graph in its filesystem at /types. Both kernel-internal types and types specified in the course of application development are organized as collections, and both kernel and application-specific collections are stored in /types/spec/c/, where c is a collection name. Each file in collection directory c is a graph node, and each file's name is the type's hash. Another directory, /types/all/, provides a single repository of type descriptions and represents all of the collections in aggregate. In its current implementation, Ethos loads the directory /types/all/ at boot time; we plan to modify Ethos so that it also reloads /types/all/ on demand when an administrator installs a new type.

We envision that application type hashes will be installed by an Ethos packaging system and will remain until the package and all of its nodes are removed. A type with hash h can be removed from /types/all/ only when it is not present in any directory /types/spec/*/ and it is not used as the type hash for any directory in the Ethos filesystem.

6.2.4 Type hash algorithm

ETN identifies types based on the hash of their syntactic and semantic specification, ensuring each type has a UUID. The possibility of cyclic types complicates this process somewhat. For example, the ETN in Figure 21a contains the cycle $V \rightarrow W \rightarrow V$. Here we describe the

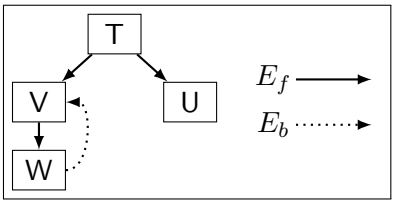


Figure 20: A type graph partitioned into E_f and E_b

```
1 T struct {                          h5 = hash(T struct { aRef1 *h2 aRef2 *h4 })
2     aRef1 *U
3     aRef2 *V
4 }

6 U struct { anInteger uint32 }       h2 = hash(U struct { anInteger uint32 })
7 V struct { leadsToACycle *W }       h1 = hash(V struct { leadsToACycle *W })
                                      h4 = hash(V struct { leadsToACycle *h3 })
9 W struct { aCyclicRef *V }          h3 = hash(W struct { aCyclicRef *h1 })
```

(a) ETN (b) Hash computation (h subscript indicates order)

Figure 21: Sample ETN structure containing the cycle $V \rightarrow W \rightarrow V$, along with its hash computation sequence

algorithm typeHash which calculates a type hash for T when given a type graph that describes T.

Let t be the type graph node corresponding to T and let $G = (N, E)$ be a directed graph. N is the set of non-primitive ETN definitions reachable from t, and E consists of the edges $[n, n']$ where node n directly references node n'.

First, typeHash's partition computes $G' = (N, E_f)$, a Directed Acyclic Graph (DAG) rooted at t and spanning G. (Given G, partition deterministically computes the same G'.) Thus E_f contains all edges excluding those that would result in a cycle. The remaining backward edges,

$E_b = E - E_f$, are those which would introduce a cycle. Figure 20 shows the partitioning of our sample type.

We next describe how typeHash propagates hashes in E_b and E_f. Every node which references another node will contain indirectly—through a series of intermediate hashes—or directly the referenced node's hash.

First, typeHash deals with the back edges using intermediary. intermediary visits back edges $e \in E_b$, calculating the hash of the parent node of e and substituting this hash in e's child node. Included in each hash is the parent node's ETN definition, including the annotations which precede it or are contained within it. These substitutions are done in an order such that the hash is always on a node which has not yet been re-written; this is ensured by noOut (n, E_f) which means that n has no outgoing edges in E_f. Thus intermediary computes hashes for all dependencies in E_b.

Next, typeHash uses collapse to compute type hashes for each $e \in E_f$, starting at the nodes which have no out edges and chaining back toward t. After computing this hash, collapse substitutes it for references to its type in other ETN definitions, removes e from E_f, and repeats until E_f is empty.

Finally, typeHash computes t's hash.

We now provide an example of how typeHash calculates the type hash for T. Initially, $E_f = \{[T, U], [T, V], [V, W]\}$. $[W, V]$ is a back edge and thus the only member of E_b.

Algorithm 1 typeHash(t)

1: $[E_f, E_b] \leftarrow$ partition(t)
2: intermediary(E_f, E_b)
3: collapse(E_f)
4: **return** hash(t)

Algorithm 2 intermediary(E_f, E_b)

1: **while** $\exists [n'', n] \in E_f \mid \text{noOut}(n, E_f)$ **do**
2: **for all** $[n, n'] \in E_b$ **do**
3: $h \leftarrow \text{hash}(n')$
4: replace references to n' in n with h
5: $E_b \leftarrow E_b - \{[n, n']\}$
6: **end for**
7: $E_f \leftarrow E_f - \{[n'', n]\}$
8: **end while**

The hash calculations are shown in Figure 21b. First, typeHash calls intermediary (E_f, E_b). The only edge that satisfies Line 1 is $[V, W]$, and the only edge that satisfies Line 2 is $[W, V]$, so intermediary calculates the intermediate hash h_1 and replaces V in W's ETN with h_1.

Next, typeHash runs collapse (E_f). This calculates the hash for U, labeled h_2, and propagates this hash to T, replacing its reference to U with h_2. Likewise, collapse hashes the definitions of W and V to compute h_3 and h_4. At each step, typeHash replaces the child's reference in the parent's ETN with a hash. Finally, typeHash computes T's type hash, h_5.

Algorithm 3 collapse(E_f)

1: **while** $\exists [n', n] \in E_f \mid \text{noOut}(n, E_f)$ **do**
2: $h \leftarrow \text{hash}(n)$
3: replace references to n in n' with h
4: $E_f \leftarrow E_f - \{[n', n]\}$
5: **end while**

```
1  TypeA struct {
2      W uint32
3      V Any
4  }
```

(a) Example ETN that defines a structure type

```
5  a ← en.TypeA {uint32(0), uint64(1)}
6  d ← syscall.OpenDirectory ("/someDir/")
7  e ← myTypes.WriteVarTypeA(d, fileName, a)
```

(b) Go code to encode an any type to a file on Ethos; the programmer uses eg2source to generate myTypes.WriteVarTypeA from the ETN definition of TypeA

```
8  d ← syscall.OpenDirectory ("/someDir/")
9  a ← myTypes.ReadVarTypeA(d, fileName)
10 switch a.V.(type) {
11     case uint64: // Of actual type uint64.
12 }
```

(c) Go code to decode an any type from a file on Ethos; the programmer uses eg2source to generate myTypes.ReadVarTypeA from the ETN definition of TypeA

Figure 22: Go code to encode/decode an any type to/from a file on Ethos

6.2.5 Sample code

6.2.5.1 Filesystem access

Figure 22 provides an example of encoding to and decoding from a file in Go. Figure 22a defines TypeA, a struct containing an integer and any. Figure 22b demonstrates how to write TypeA. Line 5 initializes a to a TypeA structure, Line 6 opens a directory, and Line 7 writes a to a file named $fileName$ in the directory. Attempting to write an ill-formed object relative to the type associated with /someDir/ would cause a runtime error.

Figure 22c provides the inverse. Here we decode to a native Go struct. Trying to decode from a file using the wrong decode function (e.g., ReadVarTypeB) would result in a runtime error.

6.2.5.2 Any types

We now describe the details of using the any type in Figure 22. The any type in TypeA must be of an actual type known to Ethos. (Not shown is the error handling at Lines 7 and 9 should the type be unknown.) Encoding an any type encodes the type hash of the actual type followed by the encoding of the actual type. Decoding an any type uses introspection to identify the actual type (Line 10). Once the actual type is determined, the application can act on it appropriately.

6.2.5.3 Remote procedure calls

Figure 23 provides an example of invoking an RPC. Figure 23a defines an example RPC interface containing a single function, Add. The programmer must also implement iAdd and iAddReply, and we list these in Figure 23b. The generated skeleton routine IHandle dispatches to these functions.

Figure 23c provides the body of an application. It opens a network connection using Ipc and initializes e and d to the returned encoder and decoder objects, respectively. These, in turn, are wrappers for Ethos' read and write system calls, and provide access to the generated RPC stub/skeleton routines. The program next invokes e's IAdd function, thereby making an RPC request. Calling d's IHandle function causes the program to wait for an incoming RPC reply to IAdd. Attempting to write an ill-formed request relative to the type associated with

```
1  I interface {
2      ['Given two 32-bit integers, respond with the sum of them.']
3      Add(i uint32, j uint32) (r uint32)
4  }
```

(a) Example ETN that defines an RPC interface

```
5  func iAdd(e *Encoder, i1 uint32, i2 uint32) {
6      e.iAddReply (i1 + i2)
7  }

9  func iAddReply(e *Encoder, i uint32) {
10     fmt.Println("Result: ", i)
11 }
```

(b) Go code to implement RPC functions on Ethos; the programmer uses eg2source to generate e.iAddReply from the ETN definition of I

```
12 e, d ← myTypes.Ipc (hostname, serviceName)
13 e.IAdd(0, 1)
14 d.HandleI(e)
```

(c) Go code to invoke the IAdd RPC and handle the response on Ethos; the programmer uses eg2source to generate myTypes.Ipc, e.IAdd, and d.IHandle from the ETN definition of I

Figure 23: Go code to invoke an RPC and handle the response on Ethos

serviceName will cause a runtime error, and Ethos will not deliver ill-formed responses to the application.

6.2.6 In brief

Etypes integrates tightly into both the Ethos OS and the programming languages used to write Ethos applications. On Ethos, programmers declare types using ETN, which supports primitive, vector, composite, and RPC types. Applications make use of these types through the use of code generators. Etypes code generators create language-specific type definitions and procedures to read and write types from/to files and IPC channels.

OS integration begins with type UUIDs, which we call type hashes. For each type, there exists a deterministic type hash which can be calculated from the type's definition. At boot time and on demand, Ethos loads a type graph, which describes the types available to the system. Ethos uses its type graph to verify all application reads and writes: Every directory in the Ethos filesystem has a type hash associated with it, and no application on Ethos is permitted to read or write an object that is ill-formed with respect to the target directory's expected type. As IPC services have corresponding filesystem nodes, this restriction also applies to IPC and network reads and writes. Thus Ethos preserves object integrity and simplifies application error handling.

6.3 Evaluation

Our evaluation focuses on the security effect of integrating Etypes with Ethos. We first consider how Etypes addresses semantic-gap attacks, addresses parsing bugs, and conserves application code. We summarize this in Table XI, and we provide additional analysis in Chapter 8 and Chapter 9 after we describe Ethos authorization. Our performance measurements include microbenchmarks of our C and Go implementations; here we compare our results to XDR/ONC RPC and JSON. Finally, we present a more realistic use case, analyzing the performance of a messaging system we wrote for Ethos. Although Ethos has security over performance as a design goal, its performance must be acceptable—we show that Etypes performs well. We evaluated the performance of Etypes using the same computers described in Chapter 4.

TABLE XI: Security features of Etypes; BP denotes a best practice shared with another OS, N denotes a novel feature, and the primary security property (as defined in Chapter 1) that benefits from each feature is noted in parenthesis

Ethos		**Comparison**
Ethos binds a type to each OS object (e.g., files or IPC connections). (P3)	N	Existing systems use weak mechanisms such as filename extensions, magic values, or port numbers to determine the type of an OS object.
Ethos either preserves object integrity on all reads and writes, or the operation fails with no side-effect. (P3)	N	Existing systems allow applications to read or write arbitrary, possibly ill-formed data. For example, OSs do not generally restrict network communication to a particular application-layer protocol, instead depending on applications to reject ill-formed requests.
Ethos discourages the use of unstructured data system-wide, instead providing convenient encoders and decoders. (P3)	N	Existing systems often pass unstructured strings; this requires parsing, which leads to semantic-gap attacks, werewolf attacks, vulnerable parsers, or injection attacks.

6.3.1 Semantic-gap attacks

Buccafurri et al. presented the Dalí attack in the context of digital signatures [105], and Jana et al. presented chameleon attacks in the context of anti-virus software [174]. In these attacks, the type is unknown and is therefore determined in an ad hoc, and sometimes erroneous, manner. Ethos provides countermeasures for both attacks. The type of each Ethos object is known (§6.2.3), every application and utility interprets an object consistently by its type, and each type has a universal meaning defined by its annotations.

Consider two types t_1 and t_2 with identical structure; they nonetheless have different hashes in Ethos due to their semantic description (§6.1.2). An Ethos application commits to a type when it reads or writes data as an Ethos object, and Ethos enforces the type it chooses. Of course, an application could erroneously swap data of type t_1 for t_2, for example, by reading data from a file of type t_1 and then writing the data to a file of type t_2. This error is unavoid-

able through structural type checking alone, but Ethos' authorization system can restrict the application so that it can only write Ethos objects of type t_1. Even in these cases, there is no possibility of encoding errors since t_1 and t_2 are identically coded.

Certificates present a particular example of the interaction between Ethos authorization and Etypes. Without Etypes, an attacker might trick a flawed program (such as a user-downloaded game) to sign a bank transfer request with a user's secret key. We discuss how Ethos integrates types into its certificate system and authorization controls in Chapter 7 and Chapter 8, respectively.

6.3.2 Parsing vs. encoding/decoding

Direct attacks on parsing are common, because parsing is complex and deals with untyped input streams, often directly generated by an attacker. Experience shows that unnecessary parsing leads to vulnerabilities. For example, the Google Chrome browser's SVG, PDF, and H.264 parsing code has contained vulnerabilities [175; 176; 177].

Parsing is also vulnerable to more subtle attacks. Jana describes the difficulty of writing parsers that prevent werewolf attacks [174]; for anti-virus, the difficulty is that virus detectors and applications can parse data differently. For example, Jana showed that many antivirus scanners will not scan a ZIP-compressed file whose valid data is preceded by a number of garbage bytes. On the other hand, the unzip utility will decompress such a file, possibly producing an executable infected with a virus. Similar difficulties are encountered when defending against Cross-site Scripting (XSS) attacks because a server's input validation must anticipate how a client browser will parse HTML [178].

Ethos takes a different approach which was inspired by applications such as qmail which avoid parsers [179]: Ethos prefers decoders, types are enforced by the OS, and for each type only a single decoder exists.

In Ethos, there are no untyped input streams—Etypes translates between external and internal representations while being subjected to a type checker. Thus applications know their input is well-typed. Furthermore, Etypes' encoders/decoders must be generated from a high-level ETN description. Only one encoder/decoder generator (eg2source) is needed per programming language, and each type has a single encoder/decoder procedure shared across all applications. Formal methods can ensure Ethos' generators are correct, removing the possibility of the direct attacks that arise with parsers. Consistent encoding and decoding removes the opportunity for the more subtle class of attacks described above. We describe in Appendix E another benefit of Etypes and eg2source: a large reduction in the lines of code used by the kernel and applications.

6.3.3 Injection attacks

In SQL- and OS-command injection attacks, an attacker influences an unstructured string that is later parsed and acted upon by another system component [180]. For example, a naïve program might receive a file path from the network and add this path to a command line sent as input to a Unix shell. An attacker could craft a path that contains the ';' character, which could cause the shell to execute arbitrary shell commands that the attacker embeds in his path string. Such injection attack vulnerabilities remain prevalent [181].

Using Etypes to read and write structured data makes it unnecessary to combine data (i.e., the path input described above) with control strings (i.e., the ';' character). On Ethos, a shell

command or SQL query would be encoded as a particular type, instead of as unstructured text. This removes the need for fragile sanitization routines and complicated character escape sequences. The facilities provided by Etypes are more general than, but resemble, prepared SQL statements [182].

6.3.4 Encoding density

The use of the type hash to identify an `any` type means that it takes 64 bytes to explicitly encode the actual type associated with an ETE. For types other than `any`, ETE's use of implicit encoding results in a minimal encoding size. There are a few exceptions, where we made decisions affecting overhead. ETE encodes NULL pointers with a single byte, and the overhead for other pointers is also a single byte. Arrays need not have their known length encoded, but tuples, strings, and dictionaries encode their length as a 32-bit value. (Choosing a 32-bit length is possible due to the integration of Etypes and Ethos—the maximum Ethos object size is $2^{32} - 1$; larger constructions can exist as a collection of objects as discussed in §6.2.3.) RPC calls contain a procedure ID as overhead, and discriminated unions contain a 32-bit tag. ETE is not 32-bit aligned; thus it can encode values of less than 32-bits more efficiently than XDR.

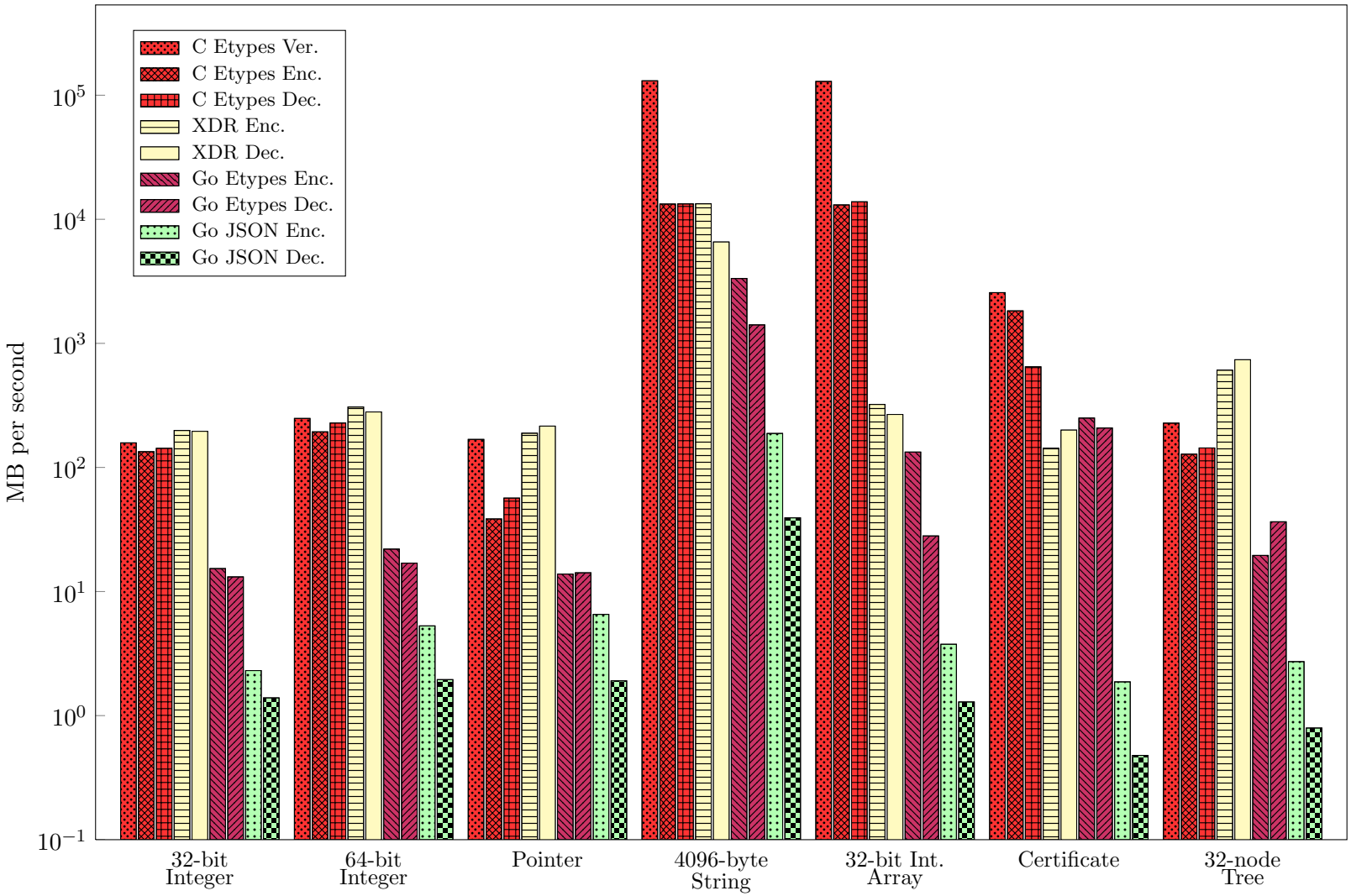


Figure 24: Etypes microbenchmarks: encode/decode to/from memory buffer

6.3.5 Performance

6.3.5.1 Microbenchmarks

We first analyzed performance by measuring the speed at which our implementation can check, encode to, and decode from a memory buffer (Figure 24). We wrote a series of microbenchmarks based on a collection of types, including primitive, vector, and composite types. We tested the speed of encoding and decoding using Etypes, XDR, and JSON. We also measured the speed of type checking. An average for each test of encoding, decoding, or checking is provided.

Figure 24 depicts our results. For scalar types, XDR is the fastest as it benefits from mandatory 32-bit alignment. XDR also encodes pointers faster than Etypes, but this is because Etypes supports cyclic and shared objects. JSON performs the slowest due to its use of runtime type introspection. Etypes' encoding of vectors containing scalar types benefits in the common case of little-endian architectures. C Etypes encoding is 0.136–22.823 (geometric mean: 1.155) times faster than XDR, and its decoding is 0.180–33.084 (geometric mean: 1.130) times faster than XDR. Verification is a common operation in the Ethos kernel, and its speed is 1.378–9.568 times faster than C Etypes encoding.

Next, we analyzed the performance of RPCs. Here each benchmark is made up of a client and server, running on a single computer. The client invokes a remote call that transfers an instance of a type to the server and the server sends it back to the client. For each type, this process is repeated a number of times. To minimize the effect of network latency, each

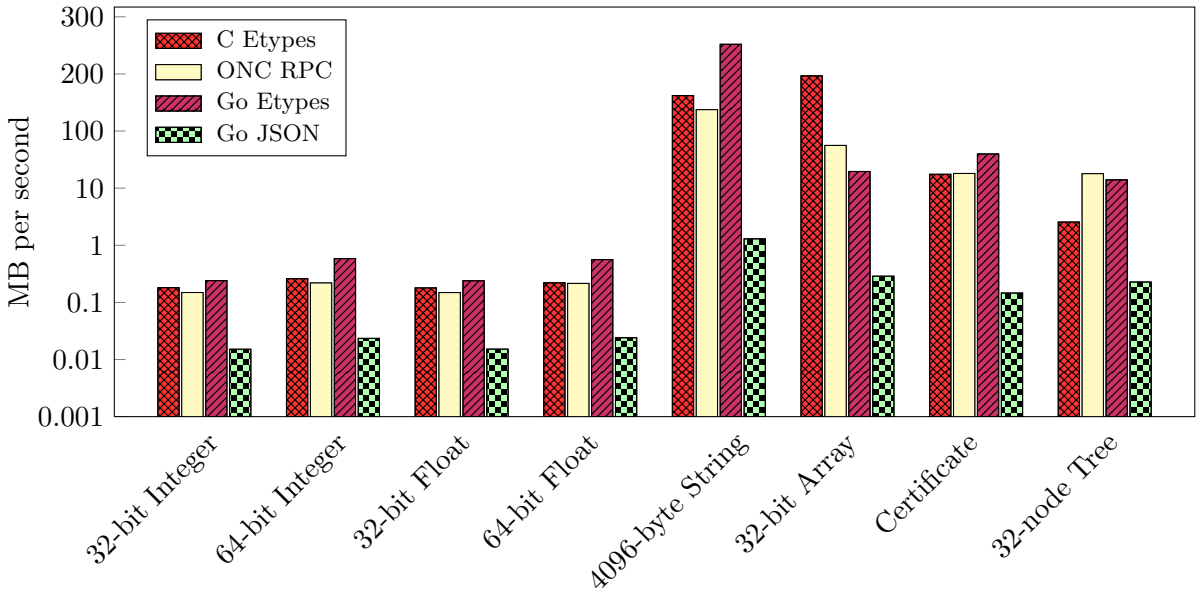


Figure 25: Etypes microbenchmarks: remote procedure calls

Etypes benchmark pipelines 20 RPCs and the ONC RPC/JSON benchmarks employ 20 client processes.

Figure 25 shows our results. JSON has the slowest performance, again due to introspection. In RPC benchmarks, the network latency becomes an important factor affecting performance. Etypes' asynchronous call model enables more efficient pipelining, reducing the effect of network latency and increasing performance. Go Etypes outperforms ONC RPC in all benchmarks except for the 32-bit integer array test and the 32-node tree. C Etypes likewise performs well: the two cases where C Etypes is outperformed by ONC RPC are the Certificate and 32-node tree. Ethos' encrypted networking stack is built on C Etypes RPCs; we presented performance measurements of this in Chapter 4.

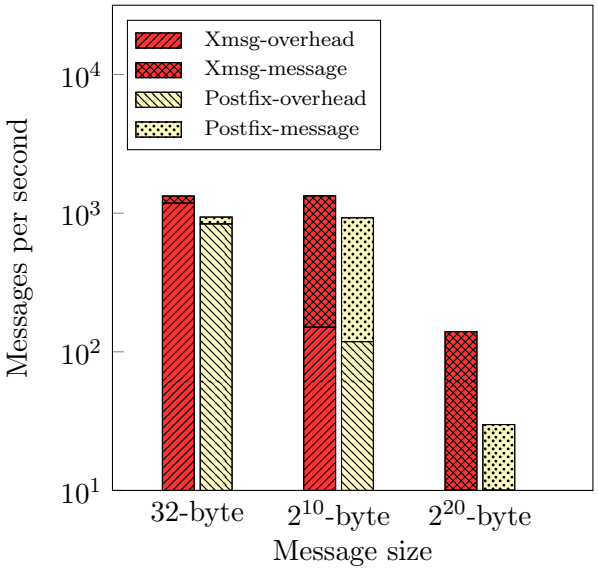


Figure 26: Performance of the eMsg messaging system

6.3.5.2 eMsg performance

To illustrate the use of Etypes, we wrote eMsg, a messaging system for Ethos. eMsg is able to send and receive a message whose type is defined in ETN and invoke RPC functions generated by eg2source.

We compare the performance of eMsg to Postfix. Since Ethos encrypts and cryptographically authenticates all network connections, we configured Postfix with TLS encryption and client certificate authentication. We wrote a client program that connects to Postfix using TLS-protected Simple Mail Transfer Protocol (SMTP) over a Unix socket and sends 2,500 emails to the server.

eMsg provides a client/server architecture roughly similar to Postfix. Both the client and server perform type-related work: the client-side kernel must check the well-formedness of

messages sent to the network and the server-side kernel must do the same for received messages, as well as check the data that the receiver writes to a user's spool directory.

Figure 26 shows the performance of eMsg and Postfix, over three different message sizes. Each bar consists of two parts: the message itself and its overhead. We modified eMsg so that it incurred an overhead roughly equal to Postfix: 1,229 bytes. (eMsg's overhead is actually smaller because it does not use Internet Message Format (IMF) headers.) For each message size, eMsg was faster than Postfix.

CHAPTER 7

CERTIFICATE SIGNING

We introduced certificates and their use for authentication in Chapter 2. Recall that a certificate is some digitally signed statement. (What is more precisely an identity certificate is often referred to as a certificate.) Certificates are a key component of robust distributed systems, because they can:

(1) allow a service reasonable assurance that a request came from a certain user and

(2) allow a user to prove that a service performed a fraudulent action without his approval.

Traditional authentication does not provide (2), so certificates are stronger than traditional authentication.

Recall that we described the requirements of a robust certificate system in Chapter 2, namely that such a system must:

(1) provide a correspondence between users and their private cryptographic keys,

(2) ensure adequate isolation of private cryptographic keys,

(3) guarantee that the system signs only what the user intends, and

(4) guarantee that the signer and relying party share a single meaning for a given certificate.

Existing systems do not sufficiently aid a user in maintaining these requirements. At issue are the many different software and hardware components involved in creating signatures. Each of these components introduces vulnerabilities.

We now describe how the design of the Ethos certificate system more strongly satisfies the requirement of non-repudiation. The use of certificates by Ethos applications adds to the implicit authentication provided by Ethos' networking system calls. Ethos certificates provide the following contributions:

- Stronger isolation properties, made possible by a sign system call. Moving the signature operation into a system call means that applications never need direct access to secret signing keys, and thus strengthens the isolation of these keys.
- A type system that allows the system to bind certificates to a fixed semantic meaning.
- An authorization framework that can restrict what a given application may sign. We call this *certificate-set authorization* and will discuss it more in Chapter 8 and Chapter 9.

The header shared by all Ethos certificates is shown in Figure 27. If a type definition begins with fields matching the certificate header, then that type defines a certificate sub-type. Each certificate contains the following fields:

Size The size of the certificate header and its payload,

Version a version field that facilitates future modifications of the certificate format,

Serial number a serial number for use with revocation,

Valid From, Valid To fields that identify the time period during which the certificate is valid,

Type hash the type hash,

Public Key the public key that will validate the certificate's signature,

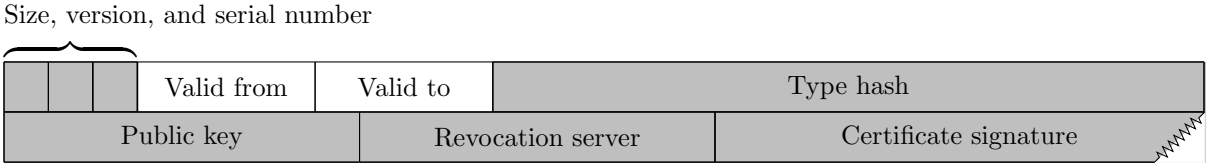


Figure 27: Ethos certificate header format; Ethos populates the fields highlighted in gray while servicing a sign system call

Revocation Server the server that maintains a certificate revocation scheme, and

Certificate Signature the certificate's digital signature.

Following the certificate header is the *certificate body* which may contain arbitrarily defined fields. While certificate headers have the same structure across all certificates, certificate bodies are specific to the type of certificate. Certificate bodies contain typed fields, and can be viewed in a manner analogous to a paper form's fields. Non-variable data is never stored in a type field; instead it is contained in the semantic description that contributes to the corresponding type ID. The semantic description describes the certificate and its fields; it is bound to a certificate sub-type as described below.

When servicing a sign system call, Ethos checks the type of the output directory, and sets the certificate's type hash to match before signing it on behalf of the user. This binds a meaning to the certificate (which might otherwise have an identical structure to another), and prevents two certificates with identical syntax but different semantics from being interchanged. Of course, a system could alternatively maintain different keys for different purposes, but eventually two seemingly separate certificates will need to be bound to the same person.

Certificate types—and the semantic description bound to them—serve to establish a predefined logic between the signer and the relying party, decreasing the semantic-level difference between the two parties. Properly designed types also reduce syntax-level differences by enforcing a canonical representation.

7.1 Interface

The sign system call takes as parameters a directory file descriptor and file name.

```
sign(dirFd, filename)
```

If the authorization policy permits the sign operation, Ethos adds the directory's type to the certificate, populates the other fields in the header (except for the validity period), signs the file, and writes it back to the filesystem.

```
verify(dirFd, filename)
```

The verification process consists of a library call that makes various system calls. Like the sign system call, the verify library call takes as parameters a directory file descriptor and file name, but verify returns a status indicating whether the certificate has a valid signature.

The library call verifies that the certificate has not expired, the signer's credentials have not been revoked, the certificate's syntax hash matches the expected type, and the digital signature on the certificate is valid. The library call also logs that the system received the certificate; at a minimum, Ethos maintains this log for the lifetime of the certificate.

7.2 In brief

Ethos is designed to provide stronger properties of certificate non-repudiation than existing systems. Applications on Ethos sign certificates by invoking the sign system call. Ethos

certificates have a consistent header format, and Ethos binds robust semantic descriptions to certificates without increasing their size. Ethos does this by adding a type hash—along with the signer's public key and the revocation server—to the certificate before signing it. Ethos also regulates the types of certificates an application may generate by enforcing certificate-set authorization.

7.3 Evaluation

Here we focus on Ethos' certificate-related security properties, which we will further evaluate along with certificate-set authorization in Chapter 9. We also compare the performance of our implementation of certificates in Ethos to OpenSSL on Linux. We were especially interested to see how a system-call-based signature would perform relative to a user-space implementation.

7.3.1 Security

Table XII summarizes the certificate-related security features provided by Ethos. Ethos isolates cryptographic keys; protects the integrity of important certificate fields; and provides a means to couple the meaning of a certificate with the certificate, reducing the semantic gap between signers and relying parties.

7.3.2 Performance

We evaluated the performance of Ethos' certificate system using the computers we described in Chapter 4. OpenSSL's `speed` utility provided the basis of our experiments. We used `speed` to measure OpenSSL's RSA implementation. Note that 1,024-bit RSA is considered unsafe today [183], and that 3,072-bit RSA is equivalent to current Ethos 128-bit security.

TABLE XII: Security features of Ethos certificates; BP denotes a best practice shared with another OS, N denotes a novel feature, and the primary security property (as defined in Chapter 1) that benefits from each feature is noted in parenthesis

Ethos		**Comparison**
The Ethos kernel naturally isolates cryptographic keys due to the use of the sign system call. (P4)	N	Typically, applications perform cryptographic signatures and thus have access to cryptographic keys. Here a compromised application could reveal a secret key.
The Ethos kernel fills in several certificate fields, most importantly ensuring the certificate's type hash and revocation server cannot be incorrectly set. (P4)	N	Typically, the contents of a certificate are set wholly by an application.
Ethos certificates have a single meaning, defined by the certificate's type. (P4)	N	Typical certificate systems are susceptible to semantic-gap attacks, because certificates are not systematically bound to a semantic meaning.

TABLE XIII: Signature measurements

System	**Signatures per second**
Ethos	7,692
OpenSSL RSA (2,048 bit key)	483
OpenSSL RSA (4,096 bit key)	74
OpenSSL DSA (2,048 bit key)	1,876
System	**Verifications per second**
Ethos	6,894
OpenSSL RSA (2,048 bit key)	18,111
OpenSSL RSA (4,096 bit key)	5,066
OpenSSL DSA (2,048 bit key)	1,517

Table XIII describes our results from comparing the signature rate of Ethos to OpenSSL. In all measurements, Ethos outperforms OpenSSL when performing signatures. We attribute this primarily to the speed of the NaCl library. Ethos verifies signatures slower than OpenSSL 2,048 RSA, but this is due to the difference between RSA and NaCl's Ed25519 algorithm (also note that NaCl provides 128-bit security, whereas 2,048 RSA is equivalent to 112-bit security). Our results demonstrate that Ethos maintains competitive performance despite moving the sign operation into a system call.

CHAPTER 8

AUTHORIZATION

Like SELinux [184], Ethos provides mandatory authorization controls which do not require that applications call a Taos-like Check procedure. Each Ethos process acts as a subject and bears an immutable user and a label. The user is inherited from the process' parent or set as a result of some call to fdSend (as described in Chapter 5), and Ethos reads the label from the process' executable in the filesystem. Objects—including files, directories, and IPC services—bear a group, owner, and a label. Ethos stores these attributes along with the object in its filesystem. Ethos uses this information while governing processes, which use system calls to access objects. In general, Ethos permissions are specified in two ways: (1) an authorization specification describes the processes which can exercise a given permission on an object, and (2) certain directory prefixes confer certain permissions on the objects they contain.

Unlike SELinux, Ethos' authorization system was designed along with its system calls, so authorization is intertwined in the semantics of Ethos' system calls. This serves to keep Ethos' authorization system simple, whereas SELinux is complex [36; 37]. In networking, this is most apparent in four properties:

(1) Ethos has fewer system calls than Linux (we discussed how Ethos reduces POSIX' sockets, pipes, etc. to import and ipc in Chapter 5).

(2) Ethos restricts applications so that they can only read, write, and sign permitted, well-formed types.

(3) Services are named textually instead of by a transport-layer port number.

(4) Ethos cryptographically authenticates the remote principal making a network connection request, and Ethos subjects such principals to its authorization policy before an application's call to import will complete and return values to user space.

Because Ethos directories contain objects of only a single type, and Ethos restricts the directories an application may write to, Ethos restricts which types an application can read, write, or sign. Such restrictions apply equally to files and application-layer protocols (i.e., the Etypes RPC an application is permitted to use). Thus Ethos restricts *what* an application may read, write, and sign whereas SELinux merely restricts *where* an application may read and write.

Ethos uses filesystem paths to name services. Passing s as the service name argument to advertise or ipc corresponds to the filesystem path /services/s/s, and the label on this path implies access controls. Unlike TCP/UDP port numbers, which have a very small name space and thus must be reused (e.g., ports 80 and 443 are used to provide very many different services), Ethos network service names are unbounded. Hence each name is one-to-one with a service. Ethos can also differentiate security levels by service name, so that, for example, Top Secret mail might use a different name than unclassified mail.

Ethos governs the way processes may interact with networking, but, due to the key differences listed above, does so in a more abstract way than SELinux. Ethos authorization affects which process may:

(1) make an outgoing connection to a service on a given remote host, where a service implies a particular well-formed application-layer protocol;

(2) accept a network request from a given remote user (whether known or anonymous) for a given service;

(3) send a file descriptor to a virtual process running a given program as a given user; and

(4) read/write file descriptors, where the data is well-formed with respect to a permitted type.

The interaction between distributors and virtual processes warrants special consideration. Ethos' authorization specification restricts virtual processes to accessing file descriptors bearing particular labels (i.e., corresponding to a certain service in the case of networking) and users. As discussed in Chapter 5, there are two patterns found in Ethos:

- Remote-user virtual processes owned by u may only read file descriptors owned by u.
- Arbitrary-user virtual processes owned by u may read file descriptors of any u'.

In the first case neither the distributor nor the virtual process (which is restricted at the system level based on the remote user) require a code audit to ensure the proper use of Ethos' security services. In the second case it is necessary to audit the distributor. The resulting work is quite modest: almost all virtual processes are expected to be remote-user, and distributors are quite

TABLE XIV: The authorization controls show how Ethos restricts each component; system administrators must audit each component as indicated (see Chapter 9)

Type	Component	Relevant authorization controls	Audit required
remote-user based	distributor	Accept network request for service	
		Send fd to virtual process	
	virtual process	Process owner = remote user	
		Commensurate access for user/program	
	client	Make outgoing connection to service	
arbitrary-user based	distributor	Accept network request for service	
		Send fd to virtual process	Confirm fdSend specifies correct recipient
	virtual process	Commensurate access for user/program	
	client	Make outgoing connection to service	

small (on the order of 100 lines). Directory permissions allow only administrators to create virtual process/distributor executables. We summarize network service audit requirements in Table XIV.

8.1 Authorizing system calls

We now discuss how the Ethos authorization policy interacts with with Ethos' system calls to fulfill its controls. Ethos integrates both Discretionary Authorization Control (DAC) and MAC in its Language for Expressing Authorization Properties (LEAP). LEAP regulates the operations subjects can invoke on objects. The LEAP permissions we will discuss here are:

c	create a file
r	read a file descriptor
w	write a file descriptor
x	execute a program
sign	sign a file
advertise	advertise/import a service
ipc	ipc to a service
fdS	send a file descriptor to a process

A LEAP unary permission has the following form, where perm is a permission; l is an object label; p is a program label; and G is a group, defined by a set of user public keys:

$$\text{perm}(l) = [p, G]$$

For example, a program labeled p, running as the user identified by the public key $pk_u \in G$ can read files labeled l:

$$\text{r}(l) = [p, G]$$

For our discussion here, we simplify the conjunction involving the user u

$$\text{perm}(l) = [p, G] \ \wedge \ pk_u \in G$$

to the more concise

$$p, u \rightarrow \text{perm}(l)$$

to mean that p, when running on behalf of u, has perm(l). In specifications, group names are written $gs.G$. To support DAC semantics a special group notation, $gs\{\$\}.G$, indicates a group specific to the owner of the object in question.

LEAP also specifies two restrictions, which resemble permissions. These are *in*, a restriction on the users who may connect to a local service, and *out*, which restricts the remote hosts that may provide a service to local programs. Thus *in* governs advertise/import and *out* governs ipc. Both of these restrictions apply to labels, but describe host groups (*out*) or user groups (*in*) instead of subjects. For example,

$$\text{in}(l) = [U]$$

restricts services labeled l so that they may be accessed only by users with public keys in U. Alternatively, an administrator may set *in* or *out* to a wildcard, thereby removing any restriction on who may connect to or provide a service. Thus administrators use wildcards to permit stranger and anonymous users to connect to services. We will now describe the permissions needed by various network-related system calls.

On the client side, an administrator may restrict to which services and hosts a given subject may connect. As we have discussed, networking-related permissions are derived from filesystem nodes, contributing to consistent authorization mechanisms. (Recall that the label on the filesystem object /services/s/s applies to the call ipc (s, host).) Within an enterprise, it may be appropriate to restrict connections to hosts that share a tightly-coupled authorization policy. Elsewhere, an administrator can choose to limit connections to trusted service providers. In

either case, applications should be permitted only to connect to the services for which they were designed, because other connections might be the result of an application compromise.

An outgoing connection to the host h for the service with label l requires:

$$p, u \rightarrow \text{ipc}(l) \ \wedge \ h \in \text{out}(l).$$

That is, to connect to a service labeled l, a program p, running as user u, must have been granted the *ipc* permission on that service. Furthermore, the server host h must be in the group of hosts permitted on the client to provide the service labeled l.

On the server side, Ethos restricts advertise and import. Ethos performs user authentication at the system layer; if import returns, Ethos has authenticated the remote user and he is authorized to connect to the given service. As discussed in Chapter 5, authorizing strangers allows even previously unknown users to maintain a unique user ID. To advertise the service labeled l and import a connection from the remote client user u_c requires:

$$\begin{aligned} \textsf{advertise:} \quad & p, u \rightarrow \text{advertise}(l) \\ \textsf{import:} \quad & p, u \rightarrow \text{advertise}(l) \ \wedge \ u_c \in \text{in}(l). \end{aligned}$$

To fdSend a file descriptor to a program labeled l requires $p, u \rightarrow \text{fdS}(l)$. The fdReceive call requires no special permission. Any virtual process may receive a file descriptor using fdReceive, but successive reads and writes are subject to access controls.

The read and peek calls require:

$$p, u \rightarrow \mathrm{r}(l).$$

The write call requires:

$$p, u \rightarrow \mathrm{w}(l).$$

The sign call requires:

$$p, u \rightarrow \mathrm{sign}(l).$$

8.2 In brief

We summarize the security features of Ethos authorization in Table XV. Ethos provides MAC, regulating the access of user-application pairs to OS objects. The system call interface that Ethos provides to applications is more abstract and provides more implicit protections than existing systems. This gives rise to a simpler, more abstract authorization policy.

Ethos can directly consider remote client users when making authorization decisions, because Ethos performs network authorization at the system level and all network users—including strangers and anonymous users—have a deterministic UUID. The authorization policies for network applications are simple; they either allow access by a set of users or further allow stranger and anonymous access. Ethos requires that network servers be implemented using either the remote-user virtual process or arbitrary-user virtual process model.

Due to the integration of Etypes, Ethos is also able to regulate the types of objects—including certificates—that applications may read and write. On Ethos, neither unauthorized

nor ill-formed network requests ever make it to an application, because both are trapped by the OS.

TABLE XV: Security features of Ethos authorization; BP denotes a best practice shared with another OS, N denotes a novel feature, and the primary security property (as defined in Chapter 1) that benefits from each feature is noted in parenthesis

Ethos		**Comparison**
Ethos provides MAC.	BP	Ethos resembles existing MAC systems because Ethos considers both the user and program running when making authorization decisions.
Ethos' simpler system calls with more abstract semantics and implicit protections give rise to a simpler, more abstract authorization policy.	N	Existing systems have complex system call interfaces and require that applications implement their own basic network protections. Existing MAC systems are cumbersome because they reflect an OS's interface design.
Ethos can directly restrict remote users because Ethos performs system-level authentication.	N	Existing systems require coordination between the system-wide authorization policy and application-level network authentication.
Ethos provides certificate-set authorization which regulates the types of certificates that an application may produce.	N	Existing systems can restrict access to cryptographic signature keys, but do not further regulate which statements applications that have access to the keys may sign.

CHAPTER 9

CUMULATIVE EVALUATION

With Ethos' authorization system defined, it is now possible to further evaluate the security properties of Ethos. We first analyze the code required for network security in existing systems (§9.1). We then analyze Ethos' individual network security properties (§9.2) and their effect on common security holes. Next, we present four case studies: remote-user virtual process (§9.3), arbitrary-user virtual process (§9.4), local authentication (§9.5), and certificate-set authorization (§9.6). Each study shows how applications benefit from Ethos' system call interface, and how the authorization of applications is simplified by this interface's implicit protections. We also analyze system administration, comparing traditional application-by-application security properties to Ethos' security settings, which apply to *all* applications (§9.7).

Ethos user space is coded in Go, but we compare mostly to POSIX C code. Although this is an apples-to-oranges comparison, two things should be kept in mind: First, in all networking code, Ethos applications require zero lines of code for many security services vs. thousands in POSIX. Second, we normalize this overhead by presenting the ratio of C authentication code to C application size.

9.1 POSIX attack surface

Application development on POSIX requires significant code and care to ensure the application safely uses networking. We discussed this in Chapter 1, noting that Dovecot contains

approximately 15,000 authentication LoC, 8% of its total C codebase; Apache's authentication modules total over 1,800 LoC; mod_ssl totals 11,703 LoC; and the third-party mod_auth_kerb contains another 1,500 LoC. Thus POSIX requires a large amount of security-sensitive application code, increasing the application attack surface. The likelihood of application-based subversion makes it more difficult for administrators to ensure that the applications they install have the requisite protections.

Experience shows that application developers cannot cope with the complexity that arises on POSIX. The US National Vulnerability Database [185] identified 699 authentication-related flaws in the last three years, 5% of the total reports. The same source reports 87 vulnerabilities in, or in the use of, OpenSSL, GnuTLS, and NSS. Studies have shown that these TLS libraries are routinely misused, including by a banking application supplied by a major US bank [11; 12].

9.2 Ethos Protections and the CWE/SANS Top 25

Ethos is differently layered: protections are transparently provided by the OS, reducing application-based subversion. Recall that we introduced protections P1–P6 in Chapter 1; these inescapable protections address several bug classes which result in security holes. Ethos defeats six error classes, or 24% of the CWE/SANS' *Top 25 Most Dangerous Software Errors* [181], and Ethos mitigates another seven error classes. Table XVI and Table XVII show how Ethos' protections address many of the Top 25 vulnerabilities.

9.3 Remote-user virtual process case study

Recall that we described resh, our remote shell utility, in Chapter 3. Here we use resh to evaluate Ethos' network security mechanisms with respect to remote-user virtual processes.

TABLE XVI: Common vulnerabilities defeated by Ethos protections

Vulnerability	**Protection**
Missing authentication for critical function	P1, P6
Use of hard-coded credentials	P1
Broken/risky cryptographic algorithm	P1, P2, P4
Allow excessive authentication attempts	P1
Use of a one-way hash without a salt	P1, P2
Missing encryption of sensitive data*	P2

*All data on Ethos is either encrypted (e.g., with network and disk writes) or isolated using memory protection

TABLE XVII: Common vulnerabilities partially mitigated by Ethos protections

Vulnerability	**Protection**
Unnecessary privileges*	P1
Incorrect permission assignment for critical resource*	P1, P4
SQL injection†	P3
OS command injection†	P3
Upload of a file with a dangerous type†	P3
Limitation of path to a restricted directory*	P6
Execution with unnecessary privileges*	P2

*Ethos' authorization system provides a consistent, system-wide way to restrict privileges
†Ethos provides a convenient mechanism for dealing with types, so it is unlikely that an application would allow communication using unstructured messages (e.g., allowing communication using strings that pass Ethos' type checks, but are parsed into something else by the application)

We compare our Ethos application to OpenSSH on POSIX. A user executes the resh client to issue commands on a remote server, in a manner similar to OpenSSH. Though simple, this application illustrates several of Ethos' advantages over POSIX.

We wrote resh in Go; it contains 547 LoC, with 383 of that in Go YACC. Resh and its corresponding server, reshDistributor/reshVP, were shown as pseudo code in Figure 10. Like

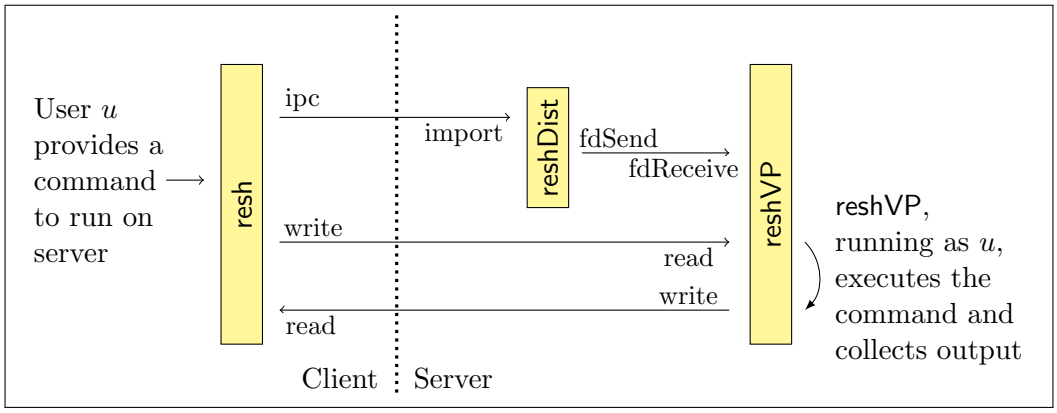


Figure 28: Interaction of the processes that make up resh

most Ethos services, resh is made up of a client, distributor, and virtual process, and we depict this architecture visually in Figure 28. For brevity, we omitted error handling code from our listings. The design pattern described here can also be used for webserver-style services.

If *Alice* is the remote user, then both resh and reshVP run as *Alice*. Ethos accomplishes this without any application encryption, authentication, or authorization code (Figure 10 has none), because Ethos provides these services at the system level. Furthermore, no application code ever has access to authentication secrets.

By default, Ethos' authorization policy disallows network-facing distributors from reading and writing a network file descriptor. Thus this distributor has two network privileges: it may advertise services and import connections of the appropriate type. Once it imports a file descriptor, it may only fdSend it to a virtual process. Only a process matching the remote user can use this descriptor. If the descriptor was sent to a virtual process running on behalf of the wrong user, Ethos authorization would prevent the process from reading or writing it. Thus

Path	**Label**
Terminal	terminal
Each program	*Program name*
/services/resh/resh	svcMsg

(a) filesystem labels; spool queues are per-user

```
1  x(resh)             = [shell, ug.out]
2  advertise(svcResh)  = [resh, ug.out]
3  out(svcResh)        =  hg.out
4  r(terminal)         = [resh, ug.out]
5  w(terminal)         = [resh, ug.out]
6  r(svcResh)          = [resh, ug.out]
7  w(svcResh)          = [resh, ug.out]

9  x(reshDist)         = [init, nobody]
10 advertise(svcResh)  = [reshDist, nobody]
11 in(svcResh)         =  ug.in
12 fdS(reshVP)         = [reshDist, nobody]

14 r(svcResh)          = [reshVP, ug{$}.in]
15 w(svcResh)          = [reshVP, ug{$}.in]
16                    ...
```

(b) LEAP specification

Figure 29: An authorization policy for resh

this distributor can only affect availability, no matter how it is written. The resulting virtual process runs as the remote user who is restricted through Ethos' authorization policy.

Figure 29 describes the authorization policy that governs resh's components, with Figure 29a showing the filesystem labels associated with resh, and Figure 29b listing a simplified version of the corresponding LEAP policy (without label or group definitions). The terminal at which a user writes and views messages bears the label terminal. Each program has a program-specific label (e.g., resh bears the label resh). We described earlier the correspondence between a service

name and filesystem node; here the node has the label svcResh. The LEAP policy references two group sets, *ug* and *hg*, respectively collections of users and hosts. Each of these group sets contains two groups, *in* and *out*, concerning incoming and outgoing connections.

First, the policy regulates resh. When run by a user in group *ug.out*, resh may connect to and read/write a remote resh service (Lines 2–6), read a command that a user enters at the terminal (Line 4), and print a command's output (Line 5). Lines 10–11 allow reshDistributor to import connections from users in *ug.in*. The distributor may then fdSend the connection to reshVP (Line12). The distributor is forbidden from reading or writing the connection. The virtual process reshVP can read or write the connection if the remote-client user matches the user running reshVP (Lines 14–15), and can perform other tasks (e.g., executing programs) depicted only with ellipses here. The Ethos type checker guarantees that all data written to or read from the resh service by either resh or reshVP is well-formed with respect to the ETN definition of the resh RPC.

OpenSSH implements its protections very carefully, requiring substantial effort [127]. OpenSSH has significant responsibilities not required of an Ethos application. Most notably, it must implement encryption and determine the remote user through some application-layer authentication protocol. Both require a substantial amount of application-specific code; OpenSSH contains nearly 12,000 lines of cryptographic- and authentication-related code, 12% of its total.

Application-based techniques must be repeated for each application and not all applications are as carefully implemented as OpenSSH. Often such POSIX applications do not execute with remote user privileges. Instead, such applications (e.g., Apache or MySQL) themselves

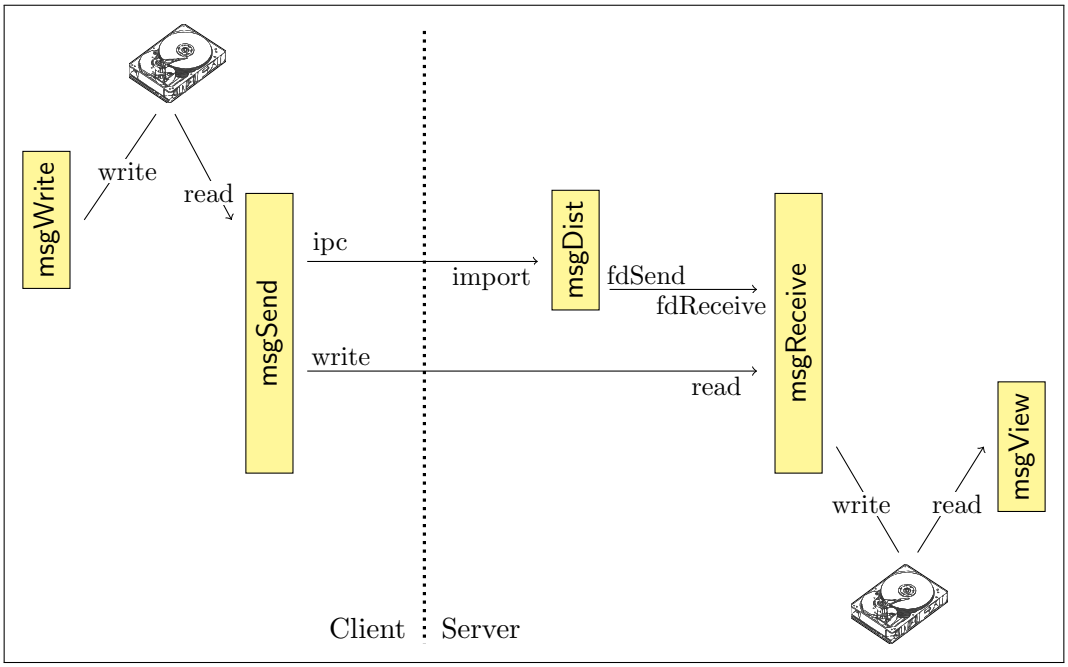


Figure 30: Interaction of the processes that make up eMsg

implement authorization controls, further adding to the application programmer's security responsibilities.

9.4 Arbitrary-user virtual process case study

We implemented on top of Ethos a networked messaging application named eMsg to demonstrate an arbitrary-user virtual process. Figure 30 visually presents the processes involved in eMsg, and Figure 31 contains a pseudo-code listing of eMsg. We wrote eMsg in Go; it contains 698 LoC, and is patterned after such Mail Delivery Agents (MDAs) as qmail and Postfix. eMsg consists of five programs, four of which pertain especially to Ethos' network protections:

msgWrite is used to compose a message and store it in the sender's outgoing spool.

msgSend is a virtual process that is either invoked by msgWrite or every ten minutes. msgSend delivers outgoing messages to a remote msgDistributor.

msgDistributor accepts messages received over the network, peeks at the recipient, and executes msgReceive with the recipient's credentials.

msgReceive runs as a virtual process with the recipient's credentials and writes incoming messages to the recipient's spool.

msgView displays the messages in a user's local incoming spool. It does not directly interact with the network, hence its listing is not included in Figure 31.

A user runs msgWrite to create a message. MsgWrite writes the message to the user's outgoing spool and notifies msgSend of the outgoing message via fdSend (Line 3). The purpose of this fdSend is to wake up the receiving process.

MsgSend calls fdReceive and beep (a timer) and then blocks until either system call completes (Line 5). In either case, msgSend reads the user's outgoing spool and attempts to send each message over the network by calling ipc and write (Lines 9–10); this communication is protected by Ethos' implicit encryption.

To receive a message, msgDistributor listens for connections. After import returns (Line 14), msgDistributor calls peek (Line 15) to obtain the recipient of the message without disturbing the stream. MsgDistributor then uses fdSend (Line 16) to pass the network file descriptor to msgReceive, a virtual process running on behalf of the recipient which writes the message to the recipient's incoming spool.

```
1  msg ← readMsgEnteredByUser()
2  writeVar("~/out/" + gettime (), msg)
3  fdSend([FdNull], getuser (), "msgSend")
```

(a) msgWrite allows user to submit a message to outgoing spool

```
4  do forever
5      wait on fdReceive() or beep(600)
6      spoolDir ← syscall.OpenDirectory ("~/out")
7      for filename in spoolDir
8          msg ← msg.ReadVarMsg(spoolDir, filename)
9          e, d ← msg.Ipc ("msg", msg.To.Host)
10         e.MsgTransmit (msg)
11         removeFile(filename)
```

(b) Client msgSend sends spooled mail after notification

```
12 listenFd ← advertise("msg")
13 do forever
14     netFd, user ← import(listenFd)
15     msg ← peek(netFd)
16     fdSend([netFd], msg.To, "msgReceive")
```

(c) Server msgDistributor sends file descriptors to msgReceive

```
17 func msgTransmit (e *Encoder, msg Msg)
18     msg.WriteVarMsg("~/in/" + gettime (), msg)

20 // Main.
21 do forever
22     fd ← FdReceive()
23     e ← etnEthos.NewWriter (fd)
24     d ← etnEthos.NewReader (fd)
25     d.HandleMsg (e)
```

(d) Server msgReceive reads messages/writes incoming spool

Figure 31: The Ethos messaging system eMsg in pseudo code

Path	**Label**
Terminal	terminal
Each program	*Program name*
/user/*user*/out/*	spoolOut
/user/*user*/in/*	spoolIn
/services/msg/msg	svcMsg

(a) filesystem labels; spool queues are per-user

```
 1  x(msgWrite)        = [shell, ug.out]
 2  r(terminal)        = [msgWrite, ug.out]
 3  c(spoolOut)        = [msgWrite, ug{$}.out]
 4  w(spoolOut)        = [msgWrite, ug{$}.out]
 5  fdS(msgSend)       = [msgWrite, ug.out]

 7  r(spoolOut)        = [msgSend, ug.out]
 8  ipc(svcMsg)        = [msgSend, ug.out]
 9  out(svcMsg)        =  hg.out
10  w(svcMsg)          = [msgSend, ug.out]

12  x(msgDist)         = [init, nobody]
13  advertise(svcMsg)  = [msgDist, nobody]
14  in(svcMsg)         =  ug.in
15  r(svcMsg)          = [msgDist, nobody]
16  fdS(msgRecv)       = [msgDist, nobody]

18  r(svcMsg)          = [msgRecv, ug.in]
19  c(spoolIn)         = [msgRecv, ug{$}.in]
20  w(spoolIn)         = [msgRecv, ug{$}.in]

22  x(msgView)         = [shell, ug.in]
23  r(spoolIn)         = [msgView, ug{$}.in]
24  w(terminal)        = [msgView, ug.in]
```

(b) LEAP specification

Figure 32: An authorization policy for eMsg

In contrast to remote-user authentication, arbitrary-user authentication allows the client to be owned by *Alice* and the virtual process msgReceive to be owned by *Bob*. This allows msgReceive to write to *Bob*'s incoming spool (and run *Bob*'s mail filters) even though the sender (i.e., remote user) is *Alice*.

Like resh, eMsg dedicates zero LoC to network authentication. It differs from resh in that authorization must be more permissive: the distributor can read messages (necessary to determine the recipient) or route a file descriptor to the wrong user. Furthermore, the msgReceive virtual process may read from any network file descriptor from the distributor, not just those that are owned by the virtual process' user, so here messages sent to the wrong user can be read by that user. Thus msgDistributor's code must be audited to ensure it will not deliver a message to the wrong recipient, but msgDistributor is very simple—35 LoC. The audit assures that it always calls fdSend with the proper recipient as the user argument. This is much less work than that required of a POSIX messaging system which implements its own authentication. For example, an audit of Postfix requires checking its use of OpenSSL and Simple Authentication and Security Layer (SASL).

Figure 32 describes the authorization policy that governs eMsg's components, with Figure 32a showing the filesystem labels associated with eMsg, and Figure 32b listing a simplified version of the corresponding LEAP policy (without label or group definitions). The labels are similar to those presented for resh, with two additions: eMsg places outgoing messages in an outgoing spool directory, labeled spoolOut; and the incoming spool directory bears the label

spoolIn. Here the service node bears the label svcMsg. We also reuse the groups ug and hg from the resh example.

First, the policy regulates msgWrite. When run by a user in group $ug.out$, msgWrite can read a message that a user enters at the terminal (Line 2), write the message to the outgoing spool (Lines 3 and 4), and invoke msgSend as a virtual process (Line 5). Here the notation $ug\{\$\}.out$ further qualifies permissions: msgWrite, running on behalf of some user u, may create or write only to directories owned by u. (Label spoolOut and $pk_u \in ug.out$ must still be satisfied.)

Lines 8–10 allow msgSend's use of ipc and write; the program can connect and write to the service labeled svcMsg if running as a user in $ug.out$. Line 9 restricts the eMsg servers msgSend may connect to to those in the group $hg.out$.

On Line 14, the policy restricts incoming connections to those originating from users in $ug.in$; such principals are cryptographically authenticated by Ethos. Of note in the remainder of the specification are Lines 19 and 20, which ensure msgReceive writes only to the recipient's own spool.

9.5 Local authentication case study

As mentioned in Chapter 5, Ethos prevents password authentication over the network. The authenticate system call cannot be used for networked communication since it requires that the password be typed on a physical, local keyboard. The local login distributor (Figure 15) is particularly restricted; it is limited to reading from stdinFd and calling fdSend. Thus the only file descriptors the login distributor may pass to fdSend are the terminal file descriptors: stdinFd, stdoutFd, and stderrFd. The distributor may prompt for a username and pass the terminal

file descriptors to the corresponding loginVP virtual process using fdSend. A compromised distributor might fdSend to the wrong user's loginVP, but that would cause the authentication to fail. Furthermore, since authenticate is a system call that does not release user secrets to user space, this breach could not be used by one user to collect the password of another. The distributor can only affect availability. This leaves loginVP which runs as the target user—it has no special privileges. The administrator needs to check that loginVP calls authenticate; however, loginVP is unique—there is only one local login program, and it is made up of fewer than 100 lines of Go and distributed with Ethos.

9.6 Certificate-set authorization case study

Users must review and approve signatures, and a system should assist them in this task. After all, a smart card may help isolate a private key but it cannot help a user decide *what* to sign. It must be possible to reason about a system and identify which application/user pair can produce which types of certificates.

In Figure 18, we presented a certificate that represented the authorization of a bank transfer. For obvious reasons, such a certificate is very sensitive. A user might expect a banking application to generate a transfer request, and he might trust that application because it was written carefully by his bank. (We already mentioned that banks in fact also produce faulty software [11], but this is an orthogonal issue.) On the other hand, the user might not equally trust a game he plays on the same computer. Obviously, the system should restrict the game so that it cannot produce a bank transfer certificate. Ethos does just this with its access controls on the sign system call. The sign call requires the *sign* permission on a given directory. Since all

of the files in a directory must conform to the directory's type, this restricts the application so that it can only sign data of permitted types. Furthermore, dangerous certificate types—such as those with machine-interpreted external references or replayable meanings—can be systematically avoided. Thus system administrators restrict the certificates that may be signed to those with vetted semantics and they do so in a centralized place, Ethos' authorization policy.

9.7 System administration

The presence of network security code in POSIX applications also has consequences for system administrators. Even small but nonetheless disparate configuration requirements—such as disabling the use of passwords by SSH—combine to create overall complexity. Performing these tasks takes inordinate time and skill [186], and is different for each application. As a result, configuration errors remain common [187]. Even the most security-sensitive organizations are unable to adequately assure their systems [8], and administrators routinely lose track of the means by which users may connect to their systems [188].

Consider configuring an application that runs on top of a web server such as Apache. An administrator must first choose an Apache authentication provider; we chose a simple one, mod_authn_file, as a lower bound on the required work. Apache also requires explicit configuration of TLS. The necessary authentication-related modules—mod_ssl, mod_auth_digest, mod_authz_host, and mod_authn_file—have 54 configuration points; one mistake can result in missing or weak encryption or authentication. As this work is performed at the application layer, the system administrator must independently configure each application (e.g., Postfix, MySQL, etc.).

For the most part, Ethos administrators can safely treat applications as black boxes—application-specific security settings are unnecessary. Furthermore, Ethos partitions general security requirements from business logic. The former is the responsibility of administrators; only the latter must be considered by application programmers. Of course, good code partitioning by application programmers will enable the system administrator to be more effective. That is, programmers should split their applications into separate processes so that they can be subjected to the Multics principle of least privilege [118].

No configuration is needed for protections P1–P6 (§9.2); they are transparently provided by Ethos. The security settings Ethos does have are centralized in the LAS/RAS and system-level authorization configuration (both are policy specifications, and so must be configurable). Authorization is external to application code, and covers network authorization by user, host, and program. Anonymity in Ethos allows universally accessible, but authenticated, service without requiring application-level user accounts.

Ethos also reduces code audit requirements. Network authentication is expected to be the common case, and this does not require any application source code audits for integrity and confidentiality. Few services require local or arbitrary-user virtual process authentication; these applications are easy to audit, especially as distributors are quite small. Distributors might also be audited for DoS protections, but this is only necessary for stranger- or anonymous-authorized services. (More complex DoS are associated with Ethos' cryptography, but that is not performed by the application.)

CHAPTER 10

CONCLUSION

10.1 Lessons learned

Many properties of current systems give rise to application-based subversion, where application-programmer errors, application-programmer omission, or application misconfiguration allows an attacker to violate a system's security policy. We found that typical network applications on POSIX require thousands of redundant lines of security-related code to provide network encryption, authentication, and authorization. We also found that configuring such applications required reading dozens of pages of documentation for each application. Thus developing and deploying network applications on POSIX is very complex, and this results in security holes.

Recent developments enable us to design much simpler, more robust systems. This is primarily due to the performance of modern hardware and the rise of virtualization. Processor speeds allow us to maximally encrypt all network messages, which in turn allows for assumptions that radically simplify system design. Virtualization means that we can provide narrower, more abstract system call interfaces, because specialized OSs can coexist with traditional OSs on the same hardware. Virtualization also aids in the deployment of a new OS.

10.2 Summary of Ethos

Ethos uses MinimaLT to provide universal network encryption, authentication, IP mobility, and DoS protections. In many cases, the latency of MinimaLT is less than with cleartext TCP/IP, and MinimaLT approaches a throughput rate of 1 Gb/s using a single CPU core. Ethos' network authentication handles known, stranger, and anonymous users.

Ethos provides strong network protections with zero lines of application security code. In contrast, each POSIX network application requires thousands of lines of repetitive, security-sensitive code. Ethos' networking system calls *implicitly* invoke encryption, authentication, type checking, and authorization, thereby shrinking application code bases and reducing their attack surface. An application programmer can neither fail to invoke nor incorrectly invoke these protections.

In Ethos, network security does not require any application-specific configuration. Instead, system administrators configure a system-wide user database and an authorization policy. This saves administrators from reading per-application security configuration documentation, configuring services for security using application-specific mechanisms, and auditing thousands of lines of code. System administrators can thus better secure applications and are less dependent on programmers for application and overall system security.

Ethos' more abstract system calls give rise to a more abstract authorization policy language. Ethos administrators use LEAP to directly restrict actions performed by remote users. This is in contrast to SELinux which requires cooperation between properly implemented application-based network authentication and the OS's access controls. Combined with Ethos' simpler

programming model and compulsory protections, Ethos' more abstract authorization controls result in systems with fewer opportunities for application-based subversion.

10.3 Directions of future research

We are beginning to shift our focus from kernel development to user space. Projects underway include writing Go packages to aid application development; designing eL, our type-safe, dynamically-checked, Etypes-integrated programming language; designing a graphics subsystem; designing a database; and developing substantial applications. We are focusing on particularly security-sensitive applications which will most stress Ethos' other security services, much like we investigated networking here. We expect that this work on applications will drive continued refinement of the system call interface that Ethos provides.

Applications will also require management tools. Thus we plan to refine our existing administrative utilities and develop new tools, such as a package manager. Ethos' type system in particular will allow for a fresh look at system administration utilities. We anticipate that eL will help us write a series of UNIX-like utilities that manipulate rich, structured data instead of text.

Work on applications will eventually allow us to certify our design and ultimately reimplement and verify our research prototype for high assurance. This will be a large project, but we have begun the initial steps of simplifying Ethos' implementation and restructuring Ethos' source code. We also plan to port Ethos so that it runs as a Hardware Virtual Machine (HVM) and port Ethos to the ARM architecture. We discuss virtualization-related considerations in Appendix D.

10.4 Application of this research

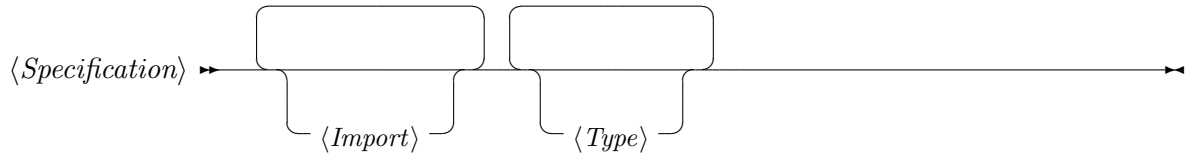
The Ethos project has produced two significant artifacts. First, we have written a usable Ethos prototype that runs on a Xen paravirtual machine, and we will soon publish a more robust implementation as open source software. This allows programmers to build real-world systems on top of Ethos. Second, we have designed a novel system call interface and its corresponding semantics. We believe that our approach of increasing robustness through simplification, as well as our resulting design, will find application in many software systems.

APPENDICES

Appendix A

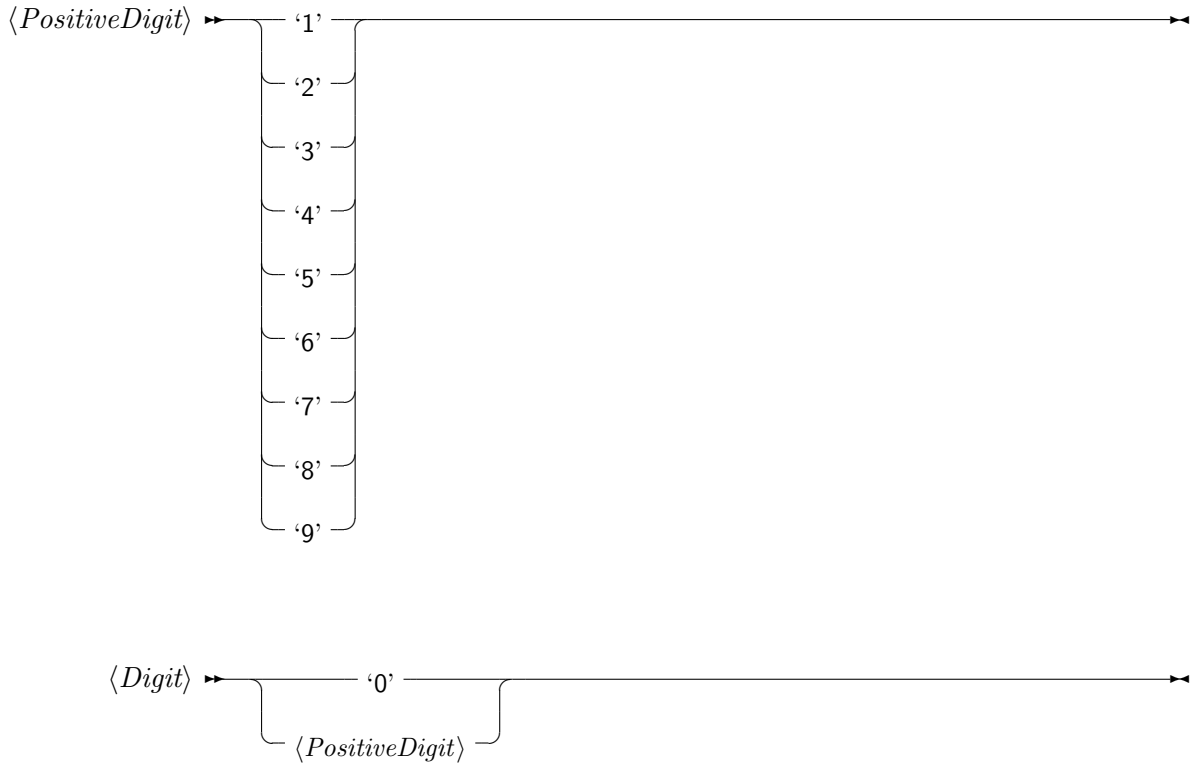
Etypes Notation

Here we present the formal grammar for ETN. A programmer defines ETN types using an ETN specification, complies the specification into a type graph, and then uses code generators to produce corresponding programming-language types and procedures (Chapter 6). An ETN specification can import an existing type graph and reference its types as it declares a new series of types. We first present the top-level rule for an ETN specification before presenting the rest of the grammar from the bottom up.

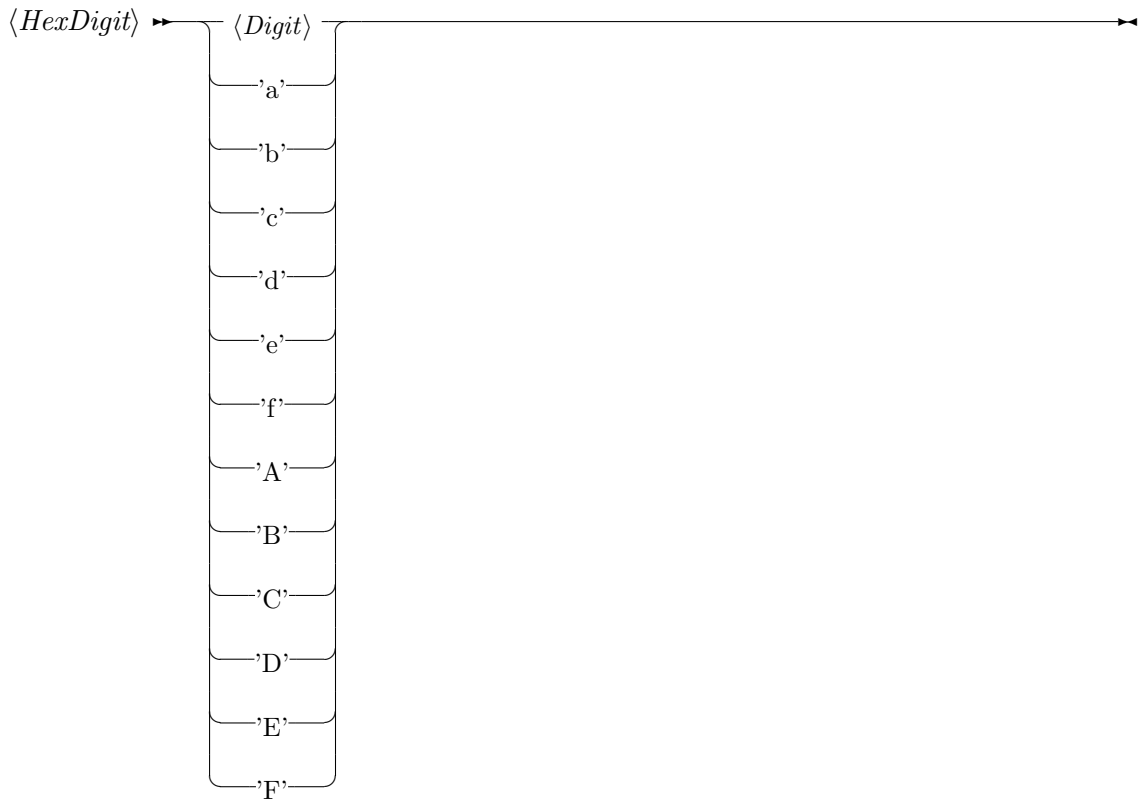


All ETN terminal symbols except for string literals are made up of letters and digits. Letters can be any Unicode code point from the general category letter (i.e., ⟨*UnicodeLetter*⟩) and underscores. String literals may further contain any Unicode characters (i.e., ⟨*UnicodeCodePoint*⟩). This allows identifiers and string literals to include a large range of characters from different languages.

Appendix A (Continued)

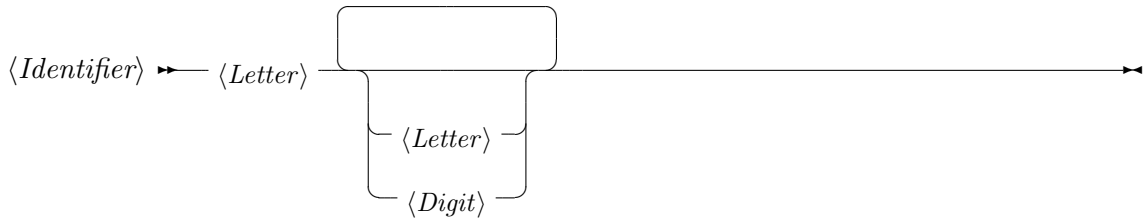


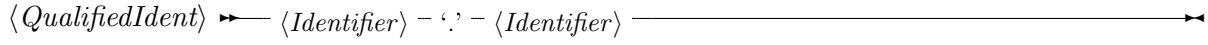
Appendix A (Continued)



Identifiers provide names for entities such as types, structure fields, and function parameters. The identifiers in a given scope must be unique, except with respect to case. ETN identifiers are case-insensitive because the capitalization of identifiers may change during code generation to satisfy programming-language requirements. A special type of identifier, the *qualified identifier*, references an entity declared elsewhere (see ⟨*Import*⟩).

Appendix A (Continued)





ETN predeclares the following identifiers, which name ETN's primitive types:

uint8 uint16 uint32 uint64 int8 int16 int32 int64 float32

float64 byte bool string any

ETN reserves the following keywords for use in constructing composite types, so they may not be used as identifiers:

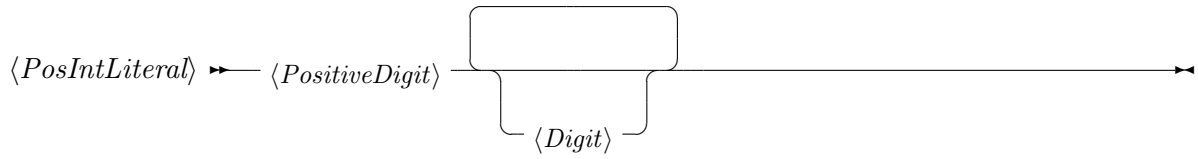
import interface struct union

ETN also forbids the use of the following characters within identifiers because they represent delimiters or otherwise special characters:

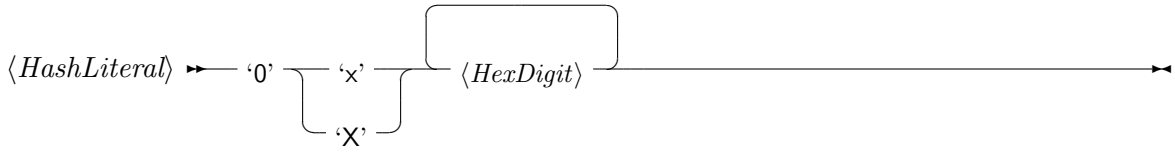
'[' ']' '{' '}' '*'

An integer literal is a sequence of decimal digits that represents an integer constant.

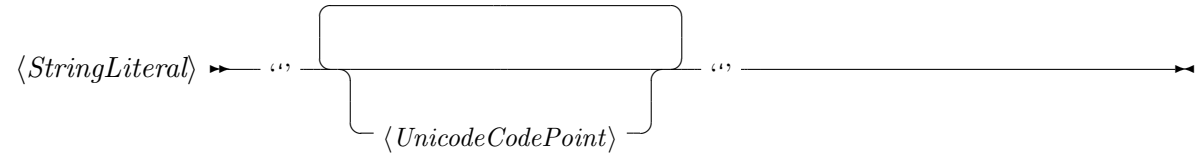
Appendix A (Continued)



A hash literal is a sequence of hexadecimal digits which is prefixed with `0x` or `0X` and represents a hash value.



A string literal is a sequence of Unicode code points representing a string constant.



ETN predeclares named instances of its primitive types (named above as predeclared identifiers and described below). Composite types—tuple, dictionary, pointer, struct, and union, and interface types—may be constructed using predeclared and user-declared type identifiers.

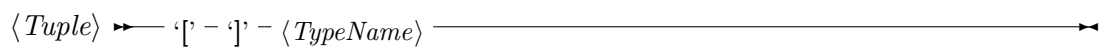
A numeric type represents a set of integer or floating-point values. ETN predeclares the following numeric types:

Appendix A (Continued)

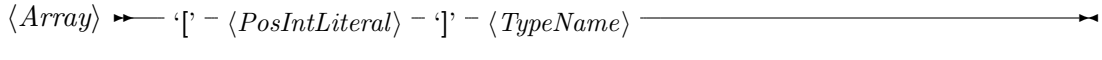
uint8	unsigned 8-bit integers (0 to $2^8 - 1$)
uint16	unsigned 16-bit integers (0 to $2^{16} - 1$)
uint32	unsigned 32-bit integers (0 to $2^{32} - 1$)
uint64	unsigned 64-bit integers (0 to $2^{64} - 1$)
int8	signed 8-bit integers (-2^7 to $2^7 - 1$)
int16	signed 16-bit integers (-2^{15} to $2^{15} - 1$)
int32	signed 32-bit integers (-2^{31} to $2^{31} - 1$)
int64	signed 64-bit integers (-2^{63} to $2^{63} - 1$)
float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers
byte	alias for uint8

ETN also provides the predeclared types bool, string, and any. The boolean type represents the truth values true and false, strings represent the set of Unicode strings, and the any type represents the set of all other declared types.

A tuple is an ordered sequence of values of a specified type but with no specified length.



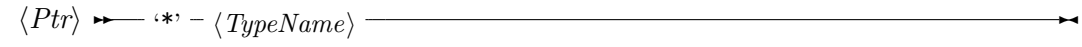
An array resembles a tuple, but specifies a positive number of elements that must be present in a valid value.



Appendix A (Continued)

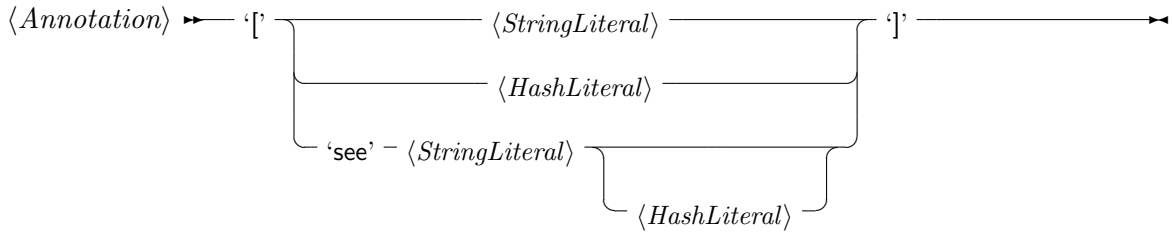
A dictionary is an unordered mapping from a set of keys of a given type to a set of values of a (possibly distinct) type. The key type must be a numeric, string, or pointer type.

A pointer type represents the set of all references to values having a given type. If a pointer value contains a reference, it is a valid reference to a value. Otherwise, the pointer contains the value nil.

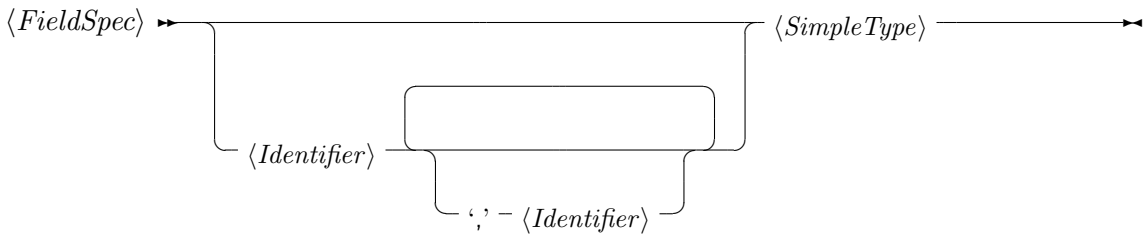


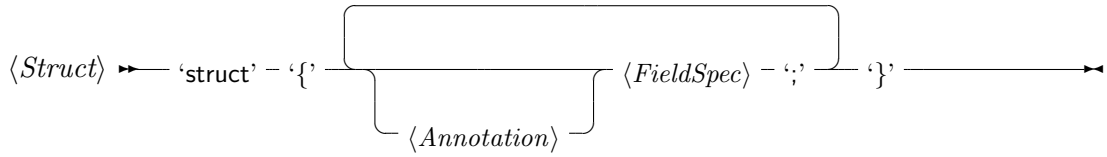
As described in Chapter 6, annotations describe the semantics of types and contribute to type hashes. Annotations may either be present in ETN as a literal or stored in an external file and referenced using see. In the former case, the use of a ⟨*StringLiteral*⟩ provides the annotation inline, and, alternatively, the use of a ⟨*HashLiteral*⟩ references a file whose hash is known (whereas the file itself might not be available). In the latter case, a ⟨*StringLiteral*⟩ represents a filesystem path and an optional ⟨*HashLiteral*⟩ ensures the contents of the referenced annotation file do not change (before the ETN is compiled into a type graph).

Appendix A (Continued)



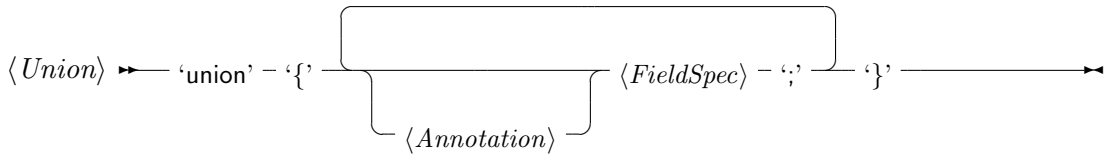
Structures define a series of fields, along with the name and type of each of these fields. Field names may be specified explicitly (i.e., in the case where ⟨*Identifier*⟩ is present before the field type) or implicitly. The body of a structure declaration forms a scope, and each field's name must be unique within this scope. Each field of a struct may be prefaced by an annotation.



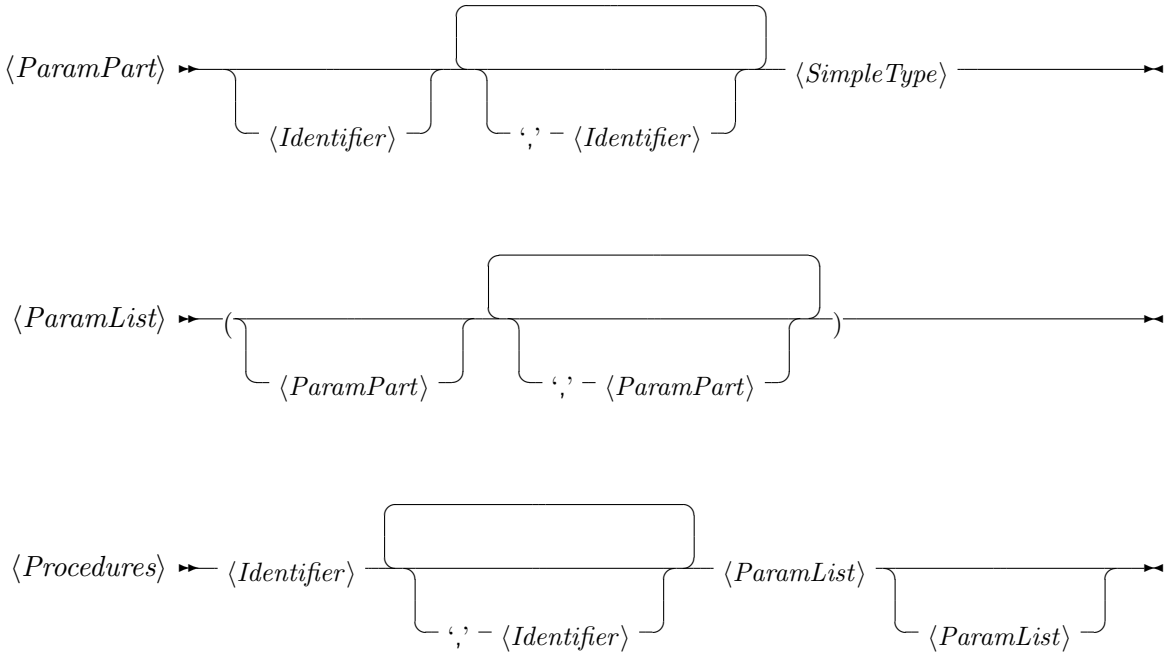


A union is specified in a manner like structures, but represents one of an explicitly defined set of types. Thus a union behaves like a restricted any type.

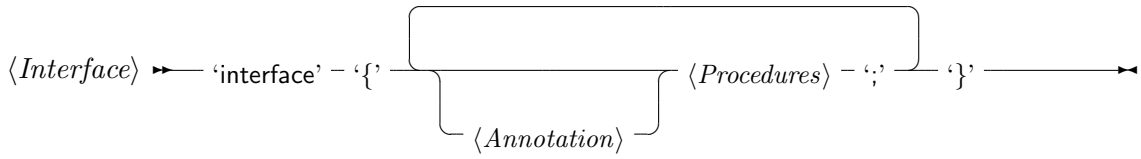
Appendix A (Continued)



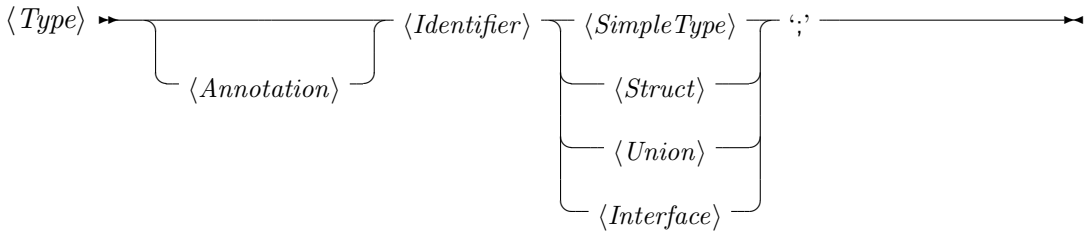
An interface specifies the collection of named procedures to which an IPC channel bearing the type responds. Each procedure in an interface has a name, a list of parameters, and an optional list of results. Within parameter or result lists, each item has a type and optional name, but the names must either all be present or all be absent. As a convenience, the ⟨*Procedures*⟩ production permits the definition of many procedures that share the same parameter and result lists.



Appendix A (Continued)

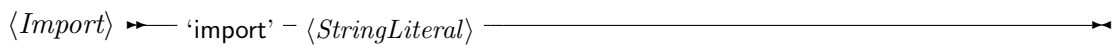


ETN types may be specified by a type name or a type literal; either composes a new type from previously declared types.



Appendix A (Continued)

Importing a type graph allows the use of the types described by it, even when their original ETN specifications are not available. After type graph G containing type t has been imported, its types can be referenced using the qualified identifier $G.t$.



ETN comments do not nest and resemble those found in C, C++, and Go. Line comments start with the character sequence // and continue up to the end of the line. Block comments start with the character sequence /∗ and continue through the character sequence ∗/.

ETN uses semicolons as terminators in a number of productions. Similar to Go, ETN type descriptions may omit semicolons in the following cases:

(1) at the end of particular lines: after the ETN lexer breaks its input into tokens, it inserts a semicolon into the token stream at the end of a non-blank line if the line's final token is not ';', ',', '(', or '{'; and

(2) before a closing '}'.

Appendix B

TYPE GRAPHS

TABLE XVIII: Base types found in Etypes Notation along with the integers that identify them within a type graph

int8	=	0	unit8	=	1	bool	=	2	int16	=	3
uint16	=	4	int32	=	5	uint32	=	6	int64	=	7
uint64	=	8	float32	=	9	float64	=	10	string	=	11
any	=	12	union	=	13	type	=	14	tuple	=	15
array	=	16	dictionary	=	17	struct	=	18	interface	=	19
method	=	20	list	=	21	pointer	=	22			

```
1  Hash []byte

3  Annotation struct {
4      Hash Hash
5      Data string
6      Base byte
7  }

9  TypeNode struct {
10     Hash Hash
11     Name string
12     Base byte
13     Annot *Annotation
14     Size uint32
15     Elems  []Hash
16     Fields []string
17 }
```

Figure 33: The types that make up a type graph

Our toolchain stores type graphs as a series of files, where each file contains one type graph node. Each node's type hash serves as the filename, and the files themselves are encoded using ETE.

To the left we list the types used to encode a type graph. Each node in the graph is of type TypeNode. Nodes have a type hash (Hash), optional name (Name), integer that identifies the type's base kind (Base), optional annotation, (Annot), optional size (Size), optional tuple of element type

Appendix B (Continued)

hashes (Elems), and optional tuple of element names (Fields). Table XVIII shows the correspondence between base kinds and their integer identifiers.

Primitive nodes describe the predeclared types supported by ETN. A primitive node may contain any of the values 0–12 in its Base field. The Hash and Name fields of a primitive node contain the type hash and predeclared identifier of its corresponding primitive type (e.g., ''int8''), respectively. All other fields are empty.

Composite nodes describe unions, tuples, arrays, dictionaries, structures, interfaces, methods, lists, or pointers. The semantics of these nodes' Size, Elems, and Fields fields vary depending on their base kind. We describe the kind-specific semantics of these nodes in Table XIX. We do not describe again the Hash, Base, or Annotation fields in this figure, because the meaning of these fields is consistent across all kinds.

TABLE XIX: Semantics of the Name, Size, Elems, and Fields fields for each kind that may be present in a type graph; we do not describe here fields that are irrelevant for a given kind or commonly defined for all kinds

Kind	Union (Base = 13)
Sample	someUnion union { a uint32; b float32 }
Name	"union"
Elems	Type hashes for the types the union may represent, in the order in which they appear in the ETN definition (e.g., the type hash of uint32 and float32)
Fields	A tuple containing a name for each field in Elems (e.g., ''a'' and ''b'')
Kind	Type (Base = 14)
Sample	someType [ ]uint32
Name	Identifier that appears in the ETN definition (e.g., ''someType'')
Elems	The type hash of the type being named (e.g., the type hash of [ ]uint32)

Appendix B (Continued)

Kind	Tuple (Base = 15)
Sample	[ ]uint32
Name	The ETN definition of the tuple (e.g., ''[ ]uint32'')
Elems	The type hash of the elements in the tuple (e.g., the type hash of uint32)
Kind	Array (Base = 16)
Sample	[8]uint32
Name	The ETN definition of the array (e.g, ''[8]uint32")
Size	Number of elements in the array (e.g., 8)
Elems	The type hash of the elements in the array (e.g., the type hash of uint32)
Kind	Dictionary (Base = 17)
Sample	[string]uint32
Name	The ETN definition of the dictionary (e.g, ''[string]uint32'')
Elems	Two type hashes: the key type and the value type (e.g., the type hash of string and uint32)
Kind	Structure (Base = 18)
Sample	someStruct struct { a uint32; b float32 }
Name	''struct''
Elems	The type hash of each structure field's type, in the order in which they appear in the ETN definition (e.g., the type hash of uint32 and float32)
Fields	A tuple containing a name for each field in Elems (e.g., ''a'' and ''b'')
Kind	Interface (Base = 19)
Sample	someInterface interface { methodB (); methodA () }
Name	''interface''
Elems	The type hash of each method of the interface, *lexicographically sorted by method name* (e.g., the type hash of methodA and methodB)
Fields	A tuple containing a name for each method in Elems (e.g., ''methodA'' and ''methodB'')

Appendix B (Continued)

Kind	Method (Base = 20)
Sample	someMethod (p1, p2 uint32) (r1 uint32)
Name	Name, as it appears in the ETN definition (e.g., "someMethod")
Elems	The hashes of two lists that describe the function parameters and result values, respectively
Fields	A tuple containing a variable name for each parameter and result value (e.g., "p1", "p2", and "r1")
Kind	Parameter list (Base = 21)
Sample	(p1, p2 uint32)
Elems	The hashes of the types in the list (e.g., three elements containing the type hash of uint32)
Kind	Pointer (Base = 22)
Sample	*uint32
Name	The ETN definition of the pointer (e.g, "*uint32")
Elems	The hash of the target type (e.g., the type hash of uint32)

Appendix C

COMPATIBILITY

Because of the huge computing infrastructure and installed software base, compatibility is essential. Compatibility impacts several levels in a system, including: applications, libraries, system calls, networking, and protocols. Ethos' clean-slate design is intended to avoid semantics that have led to errors in the past, and thus Ethos is not source-code compatible. Another means is therefore needed for real-world compatibility.

Ethos' VMM-based implementation bridges the issues of needed applications, libraries, and system calls. A computer can run several OSs concurrently, which means that the other OSs's applications are instantly available, as are its system calls and libraries. Furthermore, Ethos is directly compatible with IP and traverses Network Address Translation (NAT) networks.

By design, Ethos speaks only a single family of RPC-based application protocols. Other protocols can be bridged through the use of proxies running on other OSs. Thus we have ported Ethos's networking stack to Linux. This allows Ethos to service requests originating as any number of existing protocols. Of course, this is a less-than-ideal compromise; existing OSs will not benefit from Ethos' unified design, even as we port individual Ethos components. We believe that appropriate layering can make proxies more effective.

Collectively, these techniques allow a transition to Ethos (or another designed-for-security OS) over a period of decades, if necessary, since using Ethos does not preclude the use of any other OS. We believe that during that period, many applications will need to be rewritten to be

Appendix C (Continued)

more secure, and we hope that they will target Ethos to do so. (We note that massive rewrites and application changes were made to deal with the Y2K issue; security is a much broader issue and so its solution is likely to be more expensive.) This is a large amount of work, but we believe it to be far smaller than trying to secure critical applications on traditional OSs.

Appendix D

VIRTUALIZATION-SPECIFIC CONSIDERATIONS

Roscoe et al. used the term *disruptive virtualization* [189] when describing how virtualization could allow more expedient experimentation with OS interfaces, and we tailored some of their ideas for our own use. Writing an OS traditionally is a difficult task. This is due to the unforgiving environment when communicating directly with hardware, the idiosyncrasies found in computer architectures, the requirement for writing device drivers, and the need to complete many subsystems before the OS becomes useful. Here we discuss how we used virtualization to more expediently accomplish each of these requirements.

D.1 The appeal of a VMM-based approach

We began work on Ethos before there was stable HVM support in Xen [190]. Because of this, we chose not to build Ethos on top of an existing microkernel. Instead, we built Ethos by adding user-space processes, system calls, etc. to Xen's MiniOS. There are many known advantages to targeting a VMM when developing a new OS. This includes debugging support, profiling support, device support, and providing for backwards compatibility. We summarize each of these here (except for backwards compatibility, which we described in Chapter 1) before describing some additional tricks in §D.2.

Appendix D (Continued)

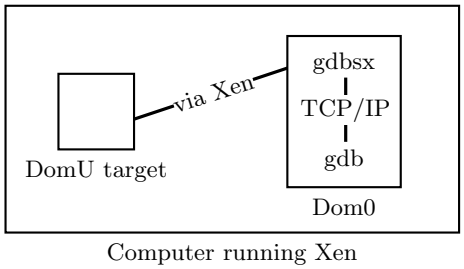


Figure 34: The Xen kernel debugging architecture

D.1.1 Debugging

The Linux kernel provides a debugging interface made up of gdb and a kernel component, kgdb. The kgdb stub consumes 5,802 LoC on Linux 3.2. Furthermore, using kgdb requires a second computer, connected to the debug target using a serial cable. Because Ethos targets Xen, it is unnecessary for us to implement special debugging support code in our kernel. Instead, we take advantage of gdbsx, a feature provided by Xen.

Figure 34 summarizes the gdbsx architecture. gdbsx is a program that runs on Dom0 that (1) communicates via Xen with the target virtual machine, and (2) communicates with gdb over a TCP/IP socket. The only special requirement from Ethos when using gdbsx is that Ethos must have been compiled into an executable with debugging information.

To automate the use of gdbsx, we wrote a program named ethosDebug that runs Ethos, gdbsx, and gdb. ethosDebug presents the user with four windows which provide Ethos' console output, terminal access to Ethos, other utility output, and the gdb interface.

D.1.2 Profiling

Profiling also benefits from the use of Xen because Xen provides a profiler, Xenoprof [191]. Like gdbsx, using Xenoprof to profile an Ethos kernel requires only that the kernel contain

Appendix D (Continued)

```
CPU: AMD64 family15h, speed 4551.44 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit ...
samples  %        image name               app name                 symbol name
1235892  40.9540  ethos.x86_32.elf         domain374-kernel         ._morebits
739271   24.4974  ethos.x86_32.elf         domain376-kernel         ._morebits
96368     3.1934  ethos.x86_32.elf         domain374-kernel         crypto...
57106     1.8923  ethos.x86_32.elf         domain376-kernel         crypto...
...
```

Figure 35: Example profile of a running Ethos kernel

debugging information. Xenoprof's support for this passive profiling has not been integrated into the mainline Linux kernel, so we occasionally port HP's patch to newer kernels and make this work available at `http://www.ethos-os.org/xenoprof`.

We use Xenoprof in conjunction with xentop, a tool that displays various statistics about running Xen domains. Figure 35 lists the partial output of a profile of two Ethos domains. Domain 374 (column 4) is a server and domain 376 is a client, and the two domains are performing a networking benchmark. We had used Xenoprof to help tune portions of the kernel to maximize network speed, and the output indicates success because the work is CPU-bound due to two cryptographic functions.

D.1.3 Device drivers

Device drivers consume a significant number of LoC in OSs. We used the utility cloc to count 5,625,090 lines of device driver code in Linux 3.2, around 50% of the total LoC in the kernel. Device driver code is often a source of bugs, including security vulnerabilities. One study found that they are a source of three to seven times more errors than general kernel code [192]. Reasons for this high bug rate include malfunctioning devices and poor documentation.

Appendix D (Continued)

Despite the large effort required, writing device drivers is often uninteresting from a research point of view.

Luckily, virtualization provides a solution. By targeting Xen, Ethos need only implement drivers for Xen's limited set of virtual devices. Ethos' network device driver is 462 LoC, and its console driver is 296 LoC. Although Ethos implements a single paravirtualized driver per device class, it is able to run on any physical device supported by Xen Dom0.

D.2 Laziness in Ethos

We make tongue-in-cheek use of the word *lazy* to describe focusing time spent while developing a new OS to maximize interesting research opportunities. Our use of lazy is similar to Larry Wall's [193]. Our focus on developing Ethos is on providing new interfaces, and so we want to work quickly through the more mundane portions of OS construction. To make progress, we chose to take shortcuts when developing Ethos' networking stack and filesystem. In addition, we tried to maximize good software engineering through the use of something we call *mixins* and deliberate testing.

D.2.1 Shadowdæmon

We wrote a Linux utility, shadowdæmon, that provides various services to an Ethos kernel running in another Xen domain. Communication between an Ethos kernel and shadowdæmon takes place over the virtual network using a series of RPCs, listed in Table XX. (We implemented Ethos-to-Ethos networking as a set of RPCs too, so this required little additional labor.) Many of the techniques below use shadowdæmon to take advantage of existing services in Dom0.

Appendix D (Continued)

TABLE XX: List of remote procedure calls supported by shadowdæmon

Name	Description
Ping	A connectivity test
GetUsers	Get the system user accounts
MetaGet	Get the metadata associated with a filesystem node
FileRead	Read the contents of the next object in a stream
FileWriteVar	Read the contents of a named file
DirectoryRemove	Remove the named directory
DirectoryCreate	Create the named directory
Random	Generate random data

D.2.2 Networking

As with most modern OSs, support for networking in Ethos is necessary. For Ethos, networking is required because we are interested in studying secure network interfaces. An Ethos host maintains two network interfaces, one to communicate with the network, and another to communicate with shadowdæmon. When Ethos boots, it reads a MAC and IP address for each of these interfaces from the command line provided by Xen. However, we were not interested in spending the time to implement the components of networking that are not interesting to our research goals. This includes Address Resolution Protocol (ARP) and a host routing table.

Linux 3.2's arp.c contains 1,000 LoC. Instead of implementing ARP and a host routing table, we engineered our Xen configuration so that both would be unnecessary in Ethos. First, we observed that if we configured Xen to route packets between its virtual and physical interfaces (instead of bridging them), then Ethos could use the same MAC address (i.e., Dom0's) for every

Appendix D (Continued)

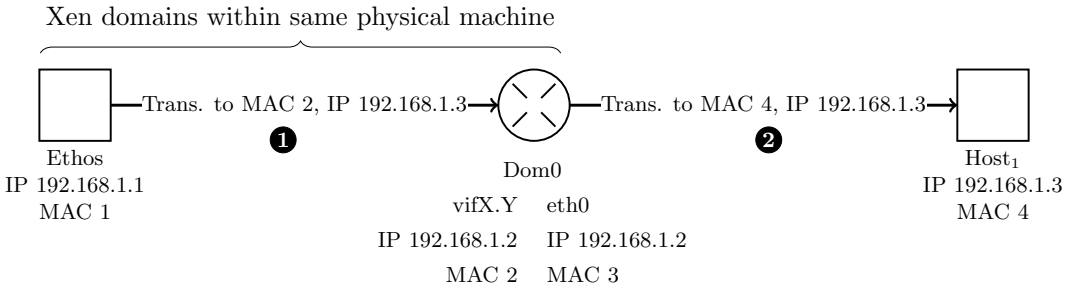
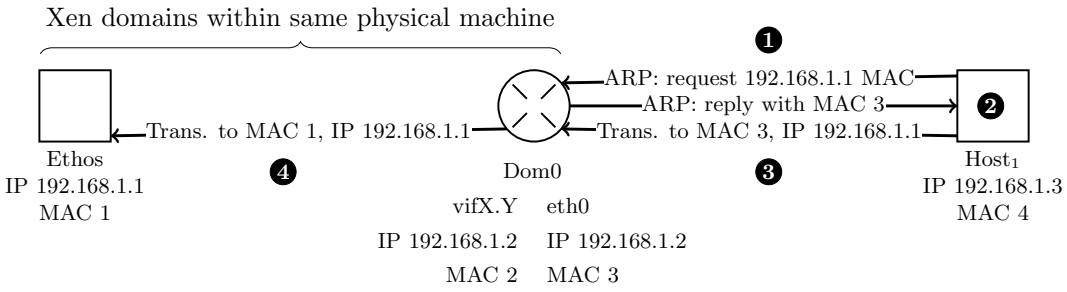


Figure 36: Sending a packet from Ethos to a remote host



(a) Sequence diagram

IP address	Interface
192.168.1.1	vifX.Y
192.168.1.0	eth0

(b) Dom0 routing Table

IP address	MAC
192.168.1.1	1

(c) Dom0 manual ARP Table

Figure 37: Using proxy ARP to deliver an incoming packet to Ethos

outgoing packet (we depict this in Figure 36). This is because in such a configuration, every packet is either destined for the Dom0 host or will be routed by the Dom0 host. The MAC Dom0 uses for its virtual network devices is known: fe:ff:ff:ff:ff:ff.

Incoming packets are slightly more complicated as is shown in Figure 37. Xen's vif-route script updates Dom0's routing tables for each new Xen domain: the script inserts a point-to-

Appendix D (Continued)

point route (37b) for each Ethos domain going out Dom0's virtual interface (e.g., vifX.Y). But each host and router on a subnet must also discover Ethos' MAC—normally hosts obtain this information using ARP. Luckily, Linux has a feature called proxy ARP which is useful here.

We configure each Dom0 virtual network interface and Dom0's primary physical network interface with the same IP address. This partitions a single subnet so that each virtual network interface on Dom0 connects to a single Ethos host, and the physical interface connects to the remainder of the subnet. We configure Dom0 to forward packets between each partition and turn on proxy ARP (sysctl flag net.ipv4.conf.all.proxy_arp). With this configuration, Dom0 will answer ARP requests on behalf of each Ethos host provided that (1) the request was received on interface n's partition, and (2) the target address belongs to a host that exists on an interface other than n. The final step is to ensure Dom0 has ARP table entries for each Ethos host. Ethos immediately sends a packet to shadowdæmon upon booting, and shadowdæmon uses this packet to update Dom0's static ARP table (Figure 37c). Thus when Dom0 receives a packet destined to an Ethos host, its routing/ARP tables allow it to deliver the packet correctly.

D.2.3 Filesystem

Ritchie and Thompson stated "*The most important job of* Unix *is to provide a file system*" [119]. It is very likely that even research OSs need one. Unfortunately, filesystems are difficult to develop—an unappealing prospect unless you are a filesystem researcher. We counted 25,829 lines in Linux 3.2's ext4 directory and 51,584 in btrfs. Previous research has shown that even production filesystems contain many bugs [194]. We have reduced this effort to shadowdæmon's 814 LoC and an Ethos component of 1,754 LoC.

Appendix D (Continued)

Instead of writing a filesystem for our Ethos prototype, we decided to take advantage of the existing filesystem on Dom0. When an Ethos application invokes a write system call, Ethos sends shadowdæmon a FileWriteVar RPC that contains a path and the contents to write. Shadowdæmon then writes the file to its local filesystem. A similar process supports an application's use of the read, fileInformation, removeFile, createDirectory, and removeDirectory system calls. The fileInformation system call is interesting in that Ethos supports file metadata typically not present on Linux. Here shadowdæmon makes use of Linux's getxattr/setxattr system calls to store Ethos metadata along with the files it describes. Shadowdæmon is also responsible for providing Ethos with random data using a Random RPC.

We store Ethos' filesystem on Dom0 at /var/lib/ethos/*domainName*. Thus this directory on Dom0 is equivalent to Ethos *domainName*'s /. We use this to our advantage; Ethos applications are presently cross-compiled using a Go compiler that runs on Linux. Installing a program is a matter of copying it to /var/lib/ethos/*domainName*/programs/. We also have all of the tools available to us in Linux when we need to initialize or repair the directories and files that make up an Ethos filesystem.

D.2.4 Mixins and libraries

We have written much of the code in Ethos in the form of libraries. Many functions useful in the kernel are also useful in user space. For example, we have a string manipulation library that is linked both into the kernel and into our user-space C library. Our abstract data type library provides another example. Some of this code is used in the Ethos kernel, Ethos user space, and Linux user space. We wanted to maximize this code reuse, but found some functions

Appendix D (Continued)

to be problematic. For example, what if a function needs to print a value (perhaps to facilitate debugging)? This is a very different operation whether taking place in the Ethos kernel or user space.

To solve this dilemma, we write mixins (named after the term's use in object-oriented programming). Mixins are small functions that perform kernel- or user-space-specific work. We have written a mixin library for the Ethos kernel, Ethos user space, and Linux user space. To solve the dilemma above, one must simply link against the appropriate mixin library. Each such library provides a different mixinPrint function, whose existence is assumed by any of our libraries that must print a value. Other mixin functions provided include mixinExit and mixinGetTimeOfDay.

Many of our libraries can be tested independently of their use in the Ethos kernel. For example, we extensively tested and profiled using gprof our allocator, libxalloc, long before we integrated it into the Ethos kernel. Furthermore, several of the libraries that we link into the Ethos kernel have unit tests we run independently of the kernel. Our RPC encoding library, libetn, has 34 unit tests and 42 benchmarks, each of which run outside of the kernel. We have found it beneficial to perform such tests outside of the kernel, because user space is a much more forgiving environment.

D.2.5 System testing

So far, we have discussed testing libraries before we link them into the kernel. Here we discuss system testing. The testing of the Ethos kernel is made easier because Ethos runs within Xen. We have written a series of tests that exercise the system calls that Ethos provides. For

Appendix D (Continued)

each test, a script runs in Dom0 that sets up the environment for the test. Then, the script boots Ethos which executes a test application. After the test application makes one or more system calls it exits, and the Dom0 script checks its results. Thus each test is automated: we generally run all 73 of our tests in series to help detect regressions during development. We have found the ability to run these focused, automated tests to be another benefit of developing on top of Xen.

D.2.6 Library operating systems

The Flux OS Toolkit (OSKit) [195] provides a reusable, low-level OS infrastructure that can serve as the basis of new research OS kernels. The idea behind OSKit is that existing modular components can be combined with novel components to produce a new OS. One interesting line of research would be to reimagine OSKit as a Xen-based augmentation to MiniOS. The obvious disadvantage of our approach to using MiniOS is that our modifications to the kernel make it difficult to merge future MiniOS work back into Ethos. Breaking MiniOS into a series of OSKit-like components would encourage additional research into custom kernels, and would allow those kernels to more easily incorporate new developments in MiniOS.

D.3 The impact of virtualization on assurance

One of the reasons we chose to target the Xen VMM over other virtualization platforms is because Xen is a bare-metal hypervisor. This is beneficial from an assurance point of view as it removes a full intermediary OS from between the VMM and hardware. However, assurance concerns do arise in the current implementation of Xen.

Appendix D (Continued)

Xen's privileged domain, Dom0, has direct access to (1) hardware I/O devices and (2) the memory of other VMs [196]. To reduce vulnerabilities due to (1), Ethos encrypts data sent to communication devices and filesystems [197]; similar techniques could secure keyboards and displays [198]. Vulnerabilities from (2) arise from DMA accesses (which use physical addresses) and because Dom0 has the ability to migrate VMs. In the former case, an I/O Memory Management Unit (IOMMU) can preserve confidentiality and integrity. In the latter case, Xen could encrypt VM pages prior to Dom0 access. Thus these security risks could be largely mitigated. An exception is availability, but unlike failures in confidentiality and integrity, availability failures are always apparent.

Another problem that arises from virtualization in general affects cryptography. Restarting a virtual machine snapshot can lead to the hosted OS using the same randomness twice, and this has been shown to render cryptographic protections vulnerable to attack [199]. Defenses to these attacks include disabling virtual machine snapshotting and using hedged cryptographic operations.

The size of Xen's code base and its exposure to Dom0 currently are impediments to building a highly secure implementation of Ethos. Orthogonal work to reduce vulnerabilities in Xen, disaggregate Dom0 [200; 201], produce alternative hypervisor technologies (e.g., Nova [202]), and verify hypervisors (e.g., MinVisor [203]) will improve the base on top of which our current research prototype is built.

Appendix E

ETHOS CODE BASE

To ensure its protections are sound, Ethos must be properly implemented. Towards this goal, the Ethos environment is engineered to remain small, even while consolidating security services. Indeed, Ethos reduces the TCB when compared to other systems by moving critical components into the OS, thereby minimizing replication. This is in contrast to minimizing the OS but then requiring much of the TCB to be implemented in applications.

As previously mentioned, targeting VMMs helps keep Ethos small, and we have also carefully chosen a small set of interfaces to export. We found further code reductions through the use of Etypes. Replacing our original hand-written RPC code in the Ethos kernel with Etypes-machine-generated code added 1,124 but removed 1,778 LoC, a net reduction of 654 LoC. More importantly, using machine-generated code reduces the hard-to-isolate bugs often resulting from RPC interface changes.

Our use of Etypes brings libetn, et2g, and eg2source into Ethos' TCB. The lines of code associated with each Etypes component are listed in Table XXI. Etypes' language support is less than 25% the size of ONC RPC, even though it supports both C and Go. Thus Etypes is small, and the protections described in Chapter 6 justify adding Etypes to the TCB.

Applications also save many lines of code through the use of ETN. For example, we observe that existing mail user, transfer, and delivery agents require an implementation of several parsers, including for SMTP message envelopes, IMF message headers, and Multipurpose In-

Appendix E (Continued)

TABLE XXI: Lines of code in Etypes (total 6,160) and ONC RPC (total 25,848)

	Component	C	Go	Template	YACC
Etypes	libetn	1,278			
	et2g		826		329
	eg2source		1,407	2,320	
ONC	libtirpc	15,105			
	rpcbind	5,264			
	rpcgen	5,479			

TABLE XXII: Lines of code in selected libcamel components

Component	Purpose	LoC
MIME	Parse MIME	13,381
SMTP	Interact using SMTP	1,487
POP3	Interact using POP3	2,958
libxml2	Parse XML/HTML	136,362

ternet Mail Extensions (MIME). Furthermore, configuration files also require parsing. These requirements increase application size. We reviewed libcamel [204], a library that implements many mail-related encoders (SMTP, POP3, MIME) and parsers (XML/HTML), and summarize its over 150,000 lines of code in Table XXII. Its encoders/decoders support communication and storage; its parsers are for languages which are used in networking for both clients and servers.

Tools such as bison and PADS [205] exist to aid in writing parsers. They clearly reduce parser costs, and they are useful on Ethos too when parsing is a necessity. However, Etypes goes further to both reduce code and provide type safety guarantees on *all* input and output.

Appendix E (Continued)

Table XXIII provides a full LoC analysis of Ethos, and Table XXIV shows the lines spent within the kernel's networking facilities. The Ethos kernel is presently made up of 37,403 LoC, excluding its cryptographic library. Of this, 1,163 lines implement fdSend, fdReceive, and authenticate. The advertise, import, and ipc system calls spend 1,502 lines, and Ethos' MinimaLT stack including MinimaLT, UDP, IP, and Ethernet spends 3,787 lines. Another 49,512 LoC come from NaCl, the cryptographic library used in Ethos, which has been carefully verified [206]. Although much smaller than Linux, our research prototype is larger than the seL4 and HiStar microkernels [207; 16].

If Ethos' design eventually proves to have sufficient strength and usability, it will be desirable to reimplement the Ethos research prototype in a way which can result in high assurance. This includes a microkernel implementation [26; 27; 28; 29; 30; 31], a minimalist VMM, and proof of correctness [207]. (As we begin this work, we may also find it time to reimplement some of the components described in Appendix D in a more traditional way.) Given that we are still working on user-space evaluation, such a reimplementation would be premature—our strategy has allowed us to focus on our research interests.

Appendix E (Continued)

TABLE XXIII: Ethos project lines of code analysis

		Lines of code				
Project	Use	C	Asm.	Go	ETN	YACC
NaCl	The NaCl cryptographic library	15,779	33,733			
Dual libraries	C libraries that are used both in the Ethos kernel and user space	18,465			150	
Kernel	Non-library C code that makes up the Ethos kernel	17,846	942			
Kernel support	Linux utilities that directly support a running Ethos kernel (e.g., shadowdæmon)	1,738				
Go port	Port of Go to Ethos*	145,980	6,169	265,938		5,115
ETN tools	ETN compiler and code generator			6,355		339
C libraries	C libraries for use in Ethos user space	1,966				
Go generic packages	Go packages for use in Linux or Ethos user space			212		
Go packages	Go packages for use in Ethos user space			671		
Programs	System programs distributed with Ethos	81	19	2,038	376	

*The GNU patch unified context diff beween our Go and Go r60.3 is 6,075 lines; many of the standard Go packages are not compatible with Ethos, but their line counts are included here

Appendix E (Continued)

TABLE XXIV: Kernel Lines of Code spent to provide network facilities

Kernel component	C lines of code
MinimaLT implementation (without NaCl)	3,787
advertise, import, and ipc system calls	1,502
authenticate system call	688
fdSend and fdReceive system calls	475
sign system call	424

AFTERWORD

I remarked during my dissertation defense that evaluating security is difficult. I think other researchers have struggled with this, primarily because security deals with the interaction of systems, policy, and—most consequentially—humans. It had occurred to me that this evaluation is subjective, and therefore required more than the typical engineering approach. We had spent a lot of time reducing Ethos to the simplest and smallest set of interfaces possible, that is, identifying the essence of what a system must do, and then designing it to do just that. I stated to my audience that we ought to begin collaborating with poets, and this was taken with a good deal of humor, given that it contrasted with my typical temperament. Following the deliberation of my committee, I was met with a single condition placed on my graduation—I must summarize my dissertation with a Haiku. Thus I close this work with:

Ritchie had it right,
but its grown far too complex.
Wise of old, clean slate.

CITED LITERATURE

1. Le Malécot, E., Hori, Y., and Sakurai, K.: Preliminary insight into distributed SSH brute force attacks. In Proceedings of the 2008 IEICE General Conference. The Institute of Electronics, Information and Communication Engineers, March 2008.

2. Belson, D.: The state of the Internet. Technical Report Q1 2011, Akamai, 2011. `http://www.akamai.com/stateoftheinternet/`.

3. Moore, T., Clayton, R., and Anderson, R.: The economics of online crime. The Journal of Economic Perspectives, 23(3):pp. 3–20, Summer 2009.

4. Kanich, C., Weavery, N., McCoy, D., Halvorson, T., Kreibichy, C., Levchenko, K., Paxson, V., Voelker, G. M., and Savage, S.: Show me the money: characterizing spam-advertised revenue. In Proceedings of the 20th USENIX Security Symposium, Berkeley, CA, USA, August 2011. USENIX Association.

5. Caballero, J., Grier, C., Kreibich, C., and Paxson, V.: Measuring Pay-per-Install: The Commoditization of Malware Distribution. In Proceedings of the the 20th USENIX Security Symposium, Berkeley, CA, USA, August 2011. USENIX Association.

6. Lynn III, W. J.: Defending a new domain. Foreign Affairs, 89(5):97–108, September 2010.

7. Kennedy, P., Takai, T., Ferguson, T., Stone, C., and Paul, K.: Information sharing in the era of WikiLeaks: Balancing security and collaboration. US Senate Homeland Security and Government Affairs Committee hearing, March 2011. `http://www.gpo.gov/fdsys/pkg/CHRG-112shrg66677/html/CHRG-112shrg66677.htm`.

8. Zorz, Z.: NSA considers its networks compromised. Help Net Security, December 2010. `http://www.net-security.org/secworld.php?id=10333` (accessed May 15, 2013).

9. Saltzer, J. H., Reed, D. P., and Clark, D. D.: End-to-end arguments in system design. ACM Transactions on Computer Systems, 2(4):277–288, November 1984.

10. The Fedora Project: Fedora crypto consolidation. http://fedoraproject.org/wiki/FedoraCryptoConsolidation. [Accessed May 15, 2013].

11. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In Proceedings of the 19th ACM Conference on Computer and Communications Security, pages 38–49, New York, NY, USA, October 2012. ACM.

12. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M.: Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In Proceedings of the 19th ACM Conference on Computer and Communications Security, pages 50–61, New York, NY, USA, October 2012. ACM.

13. NIST National Vulnerability Database: CVE-2011-0411. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-0411, March 2011. [Accessed May 15, 2013].

14. NIST National Vulnerability Database: CVE-2011-1575. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1575, May 2011. [Accessed May 15, 2013].

15. NIST National Vulnerability Database: CVE-2011-1926. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1926, May 2011. [Accessed May 15, 2013].

16. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D.: Making information flow explicit in HiStar. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA, November 2006. USENIX Association.

17. Saltzer, J. H. and Schroeder, M. D.: The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, September 1975.

18. Langley, A., Modadugu, N., and Chang, W.-T.: Overclocking SSL. Presentation at Velocity: Web Performance and Operations Conference, Santa Clara, CA, USA, June 2010. http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html (accessed May 15, 2013).

19. Bittau, A., Hamburg, M., Handley, M., Mazières, D., and Boneh, D.: The case for ubiquitous transport-level encryption. In Proceedings of the the 19th USENIX Security Symposium, Berkeley, CA, USA, August 2010. USENIX Association.

20. Bernstein, D. J. and Schwabe, P.: NEON crypto. In Workshop on Cryptographic Hardware and Embedded Systems, volume 7428 of Lecture Notes in Computer Science, pages 320–339. Springer, 2012.

21. Solworth, J. A.: Robustly secure computer systems: A new security paradigm of system discontinuity. In Proceedings of the 14th New Security Paradigms Workshop, pages 55–65, September 2007.

22. Pike, R.: System software research is irrelevant. Presentation at the Evans & Sutherland Distinguished Lecture Series, The University of Utah Department of Computer Science, USA, April 2000. `http://herpolhode.com/rob/utah2000.pdf` (accessed May 15, 2013).

23. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Information Processing Letters, 56(3):131–133, November 1995.

24. Goodin, D.: Hackers break SSL encryption used by millions of sites. The Register, September 2011. `http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/` (accessed May 15, 2013).

25. Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and Doorn, L.: vTPM: Virtualizing the trusted platform module. In Proceedings of the 15th USENIX Security Symposium, Berkeley, CA, USA, July 2006. USENIX Association.

26. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M.: Mach: A new kernel foundation for UNIX development. In Proceedings of the 1986 USENIX Summer Technical Conference, Berkeley, CA, USA, July 1986. USENIX Association.

27. Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H.: Amoeba—A distributed operating system for the 1990s. Computer, 23(5):44–53, 1990.

28. Tanenbaum, A. S. and Kaashoek, M. F.: The Amoeba microkernel. In Distributed Open Systems, eds. F. M. T. Brazier and D. Johansen, pages 11–30B. Los Alamitos, Calif., IEEE Computer Society Press, February 1994.

29. Engler, D. R., Kaashoek, M. F., and James O'Toole, J.: Exokernel: An operating system architecture for application-level resource management. In Proceedings of

the 15th Symposium on Operating Systems Principles, pages 251–266, New York, NY, USA, December 1995. ACM.

30. Tanenbaum, A. S. and Woodhull, A. S.: Operating Systems: Design and Implementation 3/e. Prentice Hall International, 2006.

31. Williams, D., Reynolds, P., Walsh, K., Sirer, E. G., and Schneider, F. B.: Device driver safety through a reference validation mechanism. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, pages 241–254, Berkeley, CA, USA, December 2008. USENIX Association.

32. Department of Defense: Trusted network interpretation. Technical Report NCSC-TG-005, US Department of Defense, 1987. `http://csrc.nist.gov/publications/secpubs/rainbow/tg005.txt` (accessed May 15, 2013).

33. Zhou, J. and Gollman, D.: A fair non-repudiation protocol. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, Washington, DC, USA, May 1996. IEEE Computer Society Press.

34. Department of Defense: Trusted computer system evaluation criteria. Technical Report DOD 5200.28–STD, US Department of Defense, 1985. `http://csrc.nist.gov/publications/secpubs/rainbow/std001.txt` (accessed May 15, 2013).

35. Dingledine, R., Mathewson, N., and Syverson, P. F.: Tor: The second-generation onion router. In Proceedings of the 13th USENIX Security Symposium, Berkeley, CA, USA, August 2004. USENIX Association.

36. Hicks, B., Rueda, S., St.Clair, L., Jaeger, T., and McDaniel, P.: A logical specification and analysis for SELinux MLS policy. In Proceedings of the 12th ACM Symposium on Access Control Models and Technologies, pages 91–100, New York, NY, USA, June 2007. ACM.

37. Nakamura, Y., Sameshima, Y., and Tabata, T.: SEEdit: SELinux security policy configuration system with higher level language. In Proceedings of the 23rd Large Installation System Administration Conference, Berkeley, CA, USA, November 2009. USENIX Association.

38. Arber, T., Cooley, D., Hirsch, S., Mahan, M., and Osterritter, J.: Network security framework: Robustness strategy. In Proceedings of the 22nd National Information

Systems Security Conference. US National Institute of Standards and Technology, October 1999.

39. George, D.: The evolution of information assurance. In Proceedings of the 21st USENIX Security Symposium, Berkeley, CA, USA, August 2012. USENIX Association. (Keynote address).

40. Schroeder, M. D.: Engineering a security kernel for Multics. In Proceedings of the 5th Symposium on Operating Systems Principles, pages 25–32, New York, NY, USA, November 1975. ACM.

41. Rushby, J.: Design and verification of secure systems. In Proceedings of the 8th Symposium on Operating Systems Principles, pages 12–21, New York, NY, USA, December 1981. ACM.

42. Lampson, B., Abadi, M., Burrows, M., and Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computing Systems, 10(4):265–310, November 1992.

43. ISO/IEC Standard 15408: Common Criteria for Information Technology Security Evaluation, version 3.1 edition, July 2009. `http://www.commoncriteriaportal.org/cc/` (accessed May 15, 2013).

44. Common Criteria certified product list, 2013. `http://www.commoncriteriaportal.org/products_OS.html` [Accessed May 15, 2013].

45. Singh, S.: The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography. New York, NY, USA, Doubleday, 1st edition, 1999.

46. Kerckhoffs, A.: La cryptographie militaire. Journal des Sciences Militaires, pages 5–38, 161–191, 1883.

47. Bernstein, D. J.: Cache-timing attacks on AES, 2004. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (accessed May 15, 2013).

48. Kocher, P. C., Jaffe, J., and Jun, B.: Differential power analysis. In Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, pages 388–397, London, UK, August 1999. Springer-Verlag.

49. Messerges, T. S., Dabbish, E. A., and Sloan, R. H.: Examining smart-card security under the threat of power analysis attacks. IEEE Transactions on Computers, 51(5):541–552, May 2002.

50. Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W.: Lest we remember: Cold boot attacks on encryption keys. In Proceedings of the 17th USENIX Security Symposium, Berkeley, CA, USA, July 2008. USENIX Association.

51. Miller, S. P., Neuman, B. C., Schiller, J. I., and Saltzer, J. H.: Kerberos authentication and authorization system. Technical report, MIT, 1987.

52. Diffie, W. and Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory, IT-22(6):644–654, November 1976.

53. Law, L., Menezes, A., Qu, M., Solinas, J., and Vanstone, S.: An efficient protocol for authenticated key agreement. Technical report, University of Waterloo, Canada, 1998.

54. Rivest, R., Shamir, A., and Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, February 1978.

55. US National Security Agency: The origination and evolution of radio traffic analysis: The World War I era. Cryptologic Quarterly, 6(1), Spring 1987. `http://www.nsa.gov/public_info/_files/cryptologic_quarterly/trafficanalysis.pdf` [Accessed May 15, 2013].

56. Walsh, J.: Dark side of the band. Wired, November 2004.

57. Ioannidis, J. and Bellovin, S. M.: Implementing pushback: Router-based defense against DDoS attacks. In Proceedings of the 9th Network and Distributed System Security Symposium, Reston, VA, USA, February 2002. The Internet Society.

58. Bernstein, D. J.: SYN flooding. USENET post on alt.security, comp.security.unix, comp.security.misc, comp.security, and comp.protocols.tcp-ip, September 1996. `http://cr.yp.to/syncookies/idea` [Accessed May 15, 2013].

59. Lemon, J.: Resisting SYN flood DoS attacks with a SYN cache. In Proceedings of the 2002 BSDCON, pages 89–97, Berkeley, CA, USA, February 2002. USENIX Association.

60. Dwork, C. and Naor, M.: Pricing via processing or combating junk mail. In Advances in Cryptology, ed. E. Brickell, volume 740 of Lecture Notes in Computer Science, pages 139–147. Springer, 1993.

61. Juels, A. and Brainard, J. G.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In Proceedings of the 1999 Network and Distributed System Security Symposium, Reston, VA, USA, February 1999. The Internet Society.

62. Anderson, R.: Security Engineering: A Guide to Dependable Distributed Systems. Wiley, 2001.

63. Hiltgen, A., Kramp, T., and Weigold, T.: Secure Internet banking authentication. IEEE Security Privacy, 4(2):21–29, March-April 2006.

64. Song, D. X., Wagner, D., and Tian, X.: Timing analysis of keystrokes and timing attacks on SSH. In Proceedings of the 10th USENIX Security Symposium, Berkeley, CA, USA, August 2001. USENIX Association.

65. Anderson, J. P.: Computer security technology planning study. Technical Report ESD-TR-73-51, US Air Force Electronics Systems Division, Air Force Systems Command, Hanscom Air Force Base, MA, USA, October 1972. `http://csrc.nist.gov/publications/history/ande72.pdf` (accessed May 15, 2013).

66. Cardelli, L.: Type systems. In The Computer Science and Engineering Handbook, ed. A. B. Tucker, chapter 97. CRC Press, 2004.

67. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, September 1998.

68. Castro, M., Costa, M., and Harris, T.: Securing software by enforcing data-flow integrity. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA, Nov 2006. USENIX Association.

69. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, December 1978.

70. Wadler, P.: Linear types can change the world! In IFIP TC 2 Working Conference on Programming Concepts and Methods, eds. M. Broy and C. Jones, pages 347–359, Sea of Galilee, Israel, 1990. North Holland.

71. Igarashi, A. and Nagira, H.: Union types for object-oriented programming. In Proceedings of the 2006 ACM symposium on Applied computing, pages 1435–1441, New York, NY, USA, 2006. ACM.

72. Herlihy, M. P. and Liskov, B.: A value transmission method for abstract data types. ACM Transactions on Programming Languages and Systems, 4(4):527–551, October 1982.

73. Birrell, A. D. and Nelson, B. J.: Implementing remote procedure calls. In Proceedings of the 9th Symposium on Operating System Principles, New York, NY, USA, October 1983. ACM.

74. Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M.: Lightweight remote procedure call. ACM Transactions on Computer Systems, 8(1):37–55, February 1990.

75. Liedtke, J.: Improving IPC by kernel design. In Proceedings of the 14th Symposium on Operating System Principles, ed. B. Liskov, pages 175–188, New York, NY, USA, December 1993. ACM.

76. Wegiel, M. and Krintz, C.: Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In Proceedings of the 2010 International Conference on Object Oriented Programming Systems Languages and Applications, pages 223–240, New York, NY, USA, 2010. ACM.

77. Python Software Foundation: Python 3.3.2 documentation, Python object serialization. `http://docs.python.org/3.3/library/pickle.html` (accessed May 15, 2013), 2010.

78. Greanier, T. M.: Flatten your objects: Discover the secrets of the Java Serialization API. JavaWorld, July 2000. `http://www.javaworld.com/jw-07-2000/jw-0714-flatten.html` (accessed May 15, 2013).

79. Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S., and Zivkovic, A.: Object serialization analysis and comparison in Java and .NET. SIGPLAN Notices, 38(8):44–54, August 2003.

80. Ramey, R.: Boost 1.48 documentation, serialization. http://www.boost.org/doc/libs/1_48_0/libs/serialization/ (accessed May 15, 2013), 2009.

81. Waldo, J.: Remote procedure calls and Java remote method invocation. IEEE Concurrency, 6(3):5–7, July 1998.

82. Eisler, M.: RFC 4506: XDR: External data representation standard. http://www.ietf.org/rfc/rfc4506.txt (accessed May 15, 2013), May 2006. Status: INFORMATIONAL.

83. Crockford, D.: RFC 4627: The application/JSON media type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt (accessed May 15, 2013), July 2006. Status: INFORMATIONAL.

84. Pike, R., Dorward, S., Griesemer, R., and Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, 13(4):277–298, oct 2005.

85. Thurlow, R.: RFC 5531: RPC: Remote procedure call protocol specification version 2. http://www.ietf.org/rfc/rfc5531.txt (accessed May 15, 2013), May 2009. Status: INFORMATIONAL.

86. Vinoski, S.: CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communications Magazine, 35(2):46–55, February 1997.

87. Crawley, S. C. and Duddy, K. R.: Improving type-safety in CORBA. In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 291–304, London, UK, September 1998. Springer-Verlag.

88. Slee, M., Agarwal, A., and Kwiatkowski, M.: Thrift: Scalable cross-language services implementation, 2007. http://thrift.apache.org/static/files/thrift-20070401.pdf [Accessed May 15, 2013].

89. Needham, R. M. and Schroeder, M. D.: Using encryption for authentication in large networks of computers. Communications of the ACM, 21(12):993–999, December 1978.

90. Steiner, J. G., Neuman, B. C., and Schiller, J. I.: Kerberos: An authentication service for open network systems. In USENIX Winter Conference Proceedings, pages 191–201, Berkeley, CA, USA, January 1988. USENIX Association.

91. Ben-Or, M., Goldreich, O., Micali, S., and Rivest, R. L.: A fair protocol for signing contracts (extended abstract). In Proceedings of the 12th Colloquium on Automata, Languages and Programming, pages 43–52, London, UK, July 1985. Springer-Verlag.

92. Soghoian, C. and Stamm, S.: Certified lies: detecting and defeating government interception attacks against SSL. In Proceedings of the 15th International Conference on Financial Cryptography and Data Security, pages 250–259, London, UK, February 2012. Springer-Verlag.

93. Leavitt, N.: Internet security under attack: The undermining of digital certificates. Computer, 44(12):17–20, December 2011.

94. Blaze, M., Feigenbaum, J., and Lacy, J.: Decentralized trust management. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, Washington, DC, USA, May 1996. IEEE Computer Society Press.

95. Rivest, R. L. and Lampson, B.: SDSI—a simple distributed security infrastucture. Technical report, MIT, April 1996. `http://people.csail.mit.edu/rivest/sdsi10.html` (accessed May 15, 2013).

96. Zimmermann, P. R.: The Official PGP Users Guide. Boston, Massachusetts, U.S.A., MIT Press, 1995.

97. Ylonen, T.: SSH—secure login connections over the Internet. In Proceedings of the 6th USENIX Security Symposium, pages 37–42, Berkeley, CA, USA, 1996. USENIX Association.

98. Cox, R., Grosse, E., Pike, R., Presotto, D., and Quinlan, S.: Security in Plan 9. In Proceedings of the 11th USENIX Security Symposium, pages 3–16, Berkeley, CA, USA, August 2002. USENIX Association.

99. Kent, G. and Shrestha, B.: Unsecured SSH—the challenge of managing SSH keys and associations. SecureIT White Paper, 2010. `http://www.secureit.com/resources/SSH_Key_Associations_Article%20Final.pdf` (accessed May 15, 2013).

100. Marchesini, J., Smith, S. W., and Zhao, M.: Keyjacking: the surprising insecurity of client-side SSL. Computers & Security, 24(2):109–123, 2005.

101. Herley, C.: So long, and no thanks for the externalities: the rational rejection of security advice by users. In Proceedings of the 2009 Workshop on New Security Paradigms Workshop, pages 133–144, New York, NY, USA, 2009. ACM.

102. Dasgupta, P., Chatha, K., and Gupta, S. K. S.: Viral attacks on the DoD common access card (CAC). http://cactus.eas.asu.edu/partha/Papers-PDF/2007/milcom.pdf (accessed May 15, 2013), 2009.

103. Arnellos, A., Lekkas, D., Spyrou, T., and Darzentas, J.: A framework for the analysis of the reliability of digital signatures for secure e-commerce. the electronic Journal for Emerging Tools & Applications, 1(4), December 2005.

104. Syverson, P.: A taxonomy of replay attacks. In Proceedings of the 7th Computer Security Foundations Workshop, pages 187–191, Washington, DC, USA, June 1994. IEEE Computer Society Press.

105. Buccafurri, F., Caminiti, G., and Lax, G.: Fortifying the Dalí attack on digital signature. In Proceedings of the 2nd International Conference on Security of Information and Networks, pages 278–287, New York, NY, USA, 2009. ACM.

106. Kain, K., Smith, S. W., and Asokan, R.: Digital signatures and electronic documents: a cautionary tale. In Proceedings of the 6th Joint Working Conference on Communications and Multimedia Security, pages 293–308, Deventer, The Netherlands, 2002. Kluwer, B.V.

107. Dierks, T. and Allen, C.: RFC 2246: The TLS protocol version 1.0. http://www.ietf.org/rfc/rfc2246.txt (accessed May 15, 2013), January 1999. Status: PROPOSED STANDARD.

108. Vratonjic, N., Freudiger, J., Bindschaedler, V., and Hubaux, J.-P.: The inconvenient truth about web certificates. In Proceedings of the 10th Workshop on the Economics of Information Security, June 2011.

109. Rescorla, E. and Modadugu, N.: RFC 6347: Datagram Transport Layer Security version 1.2. http://www.ietf.org/rfc/rfc6347.txt (accessed May 15, 2013), January 2012. Status: PROPOSED STANDARD.

110. Langley, A., Modadugu, N., and Moeller, B.: Transport Layer Security (TLS) False Start, June 2010. http://tools.ietf.org/id/draft-bmoeller-tls-falsestart-00.txt (accessed May 15, 2013).

111. Langley, A.: Transport Layer Security (TLS) Snap Start, June 2010. http://tools.ietf.org/id/draft-agl-tls-snapstart-00.txt (accessed May 15, 2013).

112. Stark, E., Huang, L.-S., Israni, D., Jackson, C., and Boneh, D.: The case for prefetching and prevalidating TLS server certificates. In Proceedings of the 19th Network and Distributed System Security Symposium, Reston, VA, USA, 2012. The Internet Society.

113. Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., and Raghavan, B.: TCP fast open. In Proceedings of the 7th International Conference on Emerging Networking Experiments and Technologies, New York, NY, USA, 2011. ACM.

114. US Committee on National Security Systems: National policy governing the use of high assurance Internet protocol encryptor. Policy No. 19, February 2007. http://www.cnss.gov/Assets/pdf/CNSSP-19.pdf (accessed May 15, 2013).

115. Stewart, R.: RFC 4960: Stream Control Transmission Protocol. http://www.ietf.org/rfc/rfc4960.txt (accessed May 15, 2013), September 2007. Status: PROPOSED STANDARD.

116. Ford, B.: Structured streams: a new transport abstraction. In Proceedings of the 2007 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 361–372, New York, NY, USA, 2007. ACM.

117. Saltzer, J. H.: Protection and the control of information sharing in Multics. Communications of the ACM, 17(7):388–402, July 1974.

118. Corbato, F. J., Saltzer, J. H., and Clingen, C. T.: Multics—the first seven years. In Proceedinfs of the 1972 Spring Joint Computer Conference, pages 571–583, New York, NY, USA, May 1972. ACM.

119. Ritchie, D. M. and Thompson, K.: The UNIX time-sharing system. Communications of the ACM, 17(7):365–375, July 1974.

120. Berger, J. L., Picciotto, J., Woodward, J. P. L., and Cummings, P. T.: Compartmented mode workstation: Prototype highlights. IEEE Transactions on Software Engineering, 16(6):608–618, 1990. Special Section on Security and Privacy.

121. Woodward, J. P. L.: Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Revision 1, The MITRE Corporation, Bedford, MA, November 1987. Also published by the Defense Intelligence Agency as document DDS-2600-5502-87.

122. Loscocco, P. and Smalley, S.: Integrating flexible support for security policies into the Linux operating system. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pages 29–42, Berkeley, CA, June 2001. The USENIX Association.

123. McKusick, K.: A narrative history of BSD. Presented at DCBSDCon, February 2009. `http://www.youtube.com/embed/ds77e3aO9nA` [Accessed May 15, 2013].

124. Jaeger, T., Butler, K., King, D. H., Hallyn, S., Latten, J., and Zhang, X.: Leveraging IPsec for mandatory access control across systems. In Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, August 2006.

125. Samar, V.: Unified login with Pluggable Authentication Modules (PAM). In Proceedings of the 3rd ACM Conference on Computer and Communications Security, New York, NY, USA, March 1996. ACM.

126. Chen, H., Wagner, D., and Dean, D.: Setuid demystified. In Proceedings of the 11th USENIX Security Symposium, Berkeley, CA, USA, August 2002. USENIX Association.

127. Provos, N., Friedl, M., and Honeyman, P.: Preventing privilege escalation. In Proceedings of the 12th USENIX Security Symposium, pages 231–242, Berkeley, CA, USA, August 2003. USENIX Association.

128. Wobber, E., Abadi, M., Burrows, M., and Lampson, B.: Authentication in the Taos operating system. In Proceedings of the 14th Symposium on Operating System Principles, pages 256–269, New York, NY, USA, 1993. ACM.

129. Wobber, T., Yumerefendi, A., Abadi, M., Birrell, A., and Simon, D. R.: Authorizing applications in Singularity. In Proceedings of the 2nd EuroSys Conference, pages 355–368, New York, NY, USA, 2007. ACM.

130. Zeldovich, N., Boyd-Wickizer, S., and Mazières, D.: Securing distributed systems with information flow control. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

131. Brumley, D. and Song, D. X.: Privtrans: Automatically partitioning programs for privilege separation. In Proceedings of the 13th USENIX Security Symposium, pages 57–72, Berkeley, CA, USA, 2004. USENIX Association.

132. Walsh, E. F.: Application of the Flask architecture to the X Window System server. In Proceedings of the 2007 SELinux Symposium. SELinux Symposium, LLC, March 2007.

133. Krohn, M. N.: Building secure high-performance web services with OKWS. In Proceedings of the 2004 USENIX Annual Technical Conference, pages 185–198, Berkeley, CA, USA, 2004. USENIX Association.

134. Bernstein, D. J.: Some thoughts on security after ten years of qmail 1.0. In Proceedings of the 2007 ACM workshop on Computer security architecture, pages 1–10, New York, NY, USA, 2007. ACM.

135. Arnold, K. and Amir, E.: Screen updating and cursor movement optimization: a library package. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1986. `http://docs.freebsd.org/44doc/psd/19.curses/paper.pdf` [Accessed May 15, 2013].

136. Solworth, J. A.: Ethos system calls. Technical report, Department of Computer Science, University of Illinois at Chicago, Chicago, IL, USA, January 2012.

137. Gummadi, P. K., Saroiu, S., and Gribble, S. D.: King: estimating latency between arbitrary Internet end hosts. In Proceedings of the 2nd Workshop on Internet Measurement, pages 5–18, New York, NY, USA, 2002. ACM.

138. Souders, S.: Velocity and the bottom line. O'Reilly Media, July 2009. `http://programming.oreilly.com/2009/07/velocity-making-your-site-fast.html` (accessed May 15, 2013).

139. Card, S. K., Robertson, G. G., and Mackinlay, J. D.: The information visualizer, an information workspace. In Proceedings of the 1991 Conference on Human Factors in Computing Systems, pages 181–188, New York, NY, USA, April 1991. ACM.

140. Eckersley, P., von Lohmann, F., and Schoen, S.: Packet forgery by ISPs: A report on the Comcast affair. Electronic Frontier Foundation, November 2007. `https://www.eff.org/files/eff_comcast_report.pdf` (accessed May 15, 2013).

141. Argyraki, K. J., Maniatis, P., Irzak, O., Ashish, S., and Shenker, S.: Loss and delay accountability for the internet. In Proceedings of the 2007 International Conference on Network Protocols, pages 194–205, Washington, DC, USA, 2007. IEEE Computer Society Press.

142. Weaver, N., Sommer, R., and Paxson, V.: Detecting forged TCP reset packets. In Proceedings of the 16th Network and Distributed Systems Security Symposium, Reston, VA, USA, February 2009. The Internet Society.

143. de Vivo, M., de Vivo, G. O., Koeneke, R., and Isern, G.: Internet vulnerabilities related to TCP/IP and T/TCP. SIGCOMM Computer Communication Review, 29(1):81–85, January 1999.

144. Keromytis, A. D., Ioannidis, S., Greenwald, M. B., and Smith, J. M.: The STRONGMAN architecture. In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, volume 1, pages 178–188, 2003.

145. Liberatore, M. and Levine, B. N.: Inferring the source of encrypted HTTP connections. In Proceedings of the 13th ACM Conference on Computer and Communications Security, pages 255–263, New York, NY, USA, October 2006. ACM.

146. Bernstein, D. J., Lange, T., and Schwabe, P.: NaCl: Networking and cryptography library. `http://nacl.cr.yp.to/` (accessed May 15, 2013).

147. Bernstein, D. J., Lange, T., and Schwabe, P.: The security impact of a new cryptographic library. In International Conference on Cryptology and Information Security in Latin America, volume 7533 of Lecture Notes in Computer Science, pages 159–176. Springer, 2012.

148. White, J. E.: A high-level framework for network-based resource sharing. In Proceedings of the 1976 National Computer Conference and Exposition, pages 561–570, New York, NY, USA, 1976. ACM.

149. Birrell, A. and Nelson, B. J.: Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1):39–59, February 1984.

150. Egevang, K. and Francis, P.: RFC 1631: The IP network address translator (NAT). `http://www.ietf.org/rfc/rfc1631.txt` (accessed May 15, 2013), May 1994. Status: INFORMATIONAL.

151. Douceur, J. R.: The Sybil attack. In Revised Papers from the First International Workshop on Peer-to-Peer Systems, pages 251–260, London, UK, 2002. Springer-Verlag.

152. Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S.: RFC 4033: DNS Security Introduction and Requirements. `http://www.ietf.org/rfc/rfc4033.txt` (accessed May 15, 2013), March 2005. Status: PROPOSED STANDARD.

153. Floyd, S.: RFC 2914: Congestion control principles. `http://www.ietf.org/rfc/rfc2914.txt` (accessed May 15, 2013), September 2000. Status: INFORMATIONAL.

154. Gettys, J. and Nichols, K.: Bufferbloat: Dark buffers in the Internet. Communications of the ACM, 55(1):57–65, January 2012.

155. Ford, B.: Directions in Internet transport evolution. IETF Journal, 3(3):29–32, December 2007.

156. Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M.: Recommendation for key management—part 1: General (revised). US National Institute of Standards and Technology, March 2007. `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf` (accessed May 15, 2013).

157. Bernstein, D. J. and Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/` (accessed May 15, 2013).

158. Shinagawa, T. and Kono, K.: Implementing a secure setuid program. In Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks, Calgary, CA, 2004. ACTA Press.

159. Bonneau, J., Herley, C., van Oorschot, P. C., and Stajano, F.: The quest to replace passwords: A framework for comparative evaluation of web authentication

schemes. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, pages 553–567, May 2012.

160. Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M.: Building verifiable trusted path on commodity x86 computers. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, May 2012.

161. Corrigan-Gibbs, H. and Ford, B.: Dissent: accountable anonymous group messaging. In Proceedings of the 17th ACM Conference on Computer and Communications Security, pages 340–350, 2010.

162. Golm, M., Felser, M., Wawersich, C., and Kleinöder, J.: The JX operating system. In Proceedings of the 2002 USENIX Annual Technical Conference, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.

163. Liu, J., George, M. D., Vikram, K., Qi, X., Waye, L., and Myers, A. C.: Fabric: a platform for secure distributed computation and storage. In Proceedings of the 22nd Symposium on Operating System Principles, New York, NY, USA, October 2009. ACM.

164. Hunt, G. C. and Larus, J. R.: Singularity: Rethinking the software stack. SIGOPS Operating Systems Review, 41(2):37–49, April 2007.

165. Greenblatt, R. D., Knight, T. F., Holloway, J. T., and Moon, D. A.: A LISP machine. In Proceedings of the 5th workshop on Computer Architecture for Non-Numeric Processing, pages 137–138, New York, NY, USA, 1980. ACM.

166. Hallgren, T., Jones, M. P., Leslie, R., and Tolmach, A.: A principled approach to operating system construction in Haskell. In Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, pages 116–128, New York, NY, USA, 2005. ACM.

167. Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C.: Extensibility, safety and performance in the SPIN operating system. In Proceedings of the 15th Symposium on Operating System Principles, pages 267–284, New York, NY, USA, December 1995. ACM.

168. Grimm, R. and Bershad, B. N.: Separating access control policy, enforcement, and functionality in extensible systems. ACM Transactions on Computing Systems, 19(1):36–70, February 2001.

169. Ganger, G. R., Engler, D. R., Kaashoek, M. F., Briceño, H. M., Hunt, R., and Pinckney, T.: Fast and flexible application-level networking on exokernel systems. ACM Transactions on Computing Systems, 20(1):49–83, February 2002.

170. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J.: Unikernels: library operating systems for the cloud. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 461–472, New York, NY, USA, March 2013. ACM.

171. Wang, R., Chen, S., and Wang, X.: Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, pages 365–379, Washington, DC, USA, May 2012. IEEE Computer Society Press.

172. Liskov, B. and Scheifler, R.: Guardians and actions: Linguistic support for robust, distributed programs. Transactions on Programming Languages and Systems, 5(3):381–404, July 1983.

173. Pike, R. and Weinberger, P.: The hideous name. In USENIX Summer Conference Proceedings, pages 563–568, Portland, Oregon, USA, June 1985.

174. Jana, S. and Shmatikov, V.: Abusing file processing in malware detectors for fun and profit. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, Washington, DC, USA, May 2012. IEEE Computer Society Press.

175. NIST National Vulnerability Database: CVE-2011-3908. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3908`, December 2011. [Accessed May 15, 2013].

176. NIST National Vulnerability Database: CVE-2011-3906. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3906`, December 2011. [Accessed May 15, 2013].

177. NIST National Vulnerability Database: CVE-2011-3025. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3025`, February 2012. [Accessed May 15, 2013].

178. Bisht, P. and Venkatakrishnan, V.: XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In Proceedings of the 5th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Springer-Verlag, 2008.

179. Bernstein, D. J.: The qmail security guarantee. `http://cr.yp.to/qmail/guarantee.html` [Accessed May 15, 2013].

180. Su, Z. and Wassermann, G.: The essence of command injection attacks in Web applications. ACM SIGPLAN Notices, 41(1):372–382, January 2006.

181. Martin, B., Brown, M., Paller, A., and Kirby, D.: Top 25 most dangerous software errors. CWE/SANS, March 2010. `http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.html` [Accessed May 15, 2013].

182. Bisht, P., Sistla, A. P., and Venkatakrishnan, V. N.: TAPS: automatically preparing safe SQL queries. In Proceedings of the 17th ACM Conference on Computer and Communications Security, pages 645–647, New York, NY, USA, October 2010. ACM.

183. Wikipedia: Key size. `http://en.wikipedia.org/wiki/Key_size` (accessed May 15, 2013), 2011.

184. Loscocco, P. and Smalley, S.: Meeting critical security objectives with Security-Enhanced Linux. In Proceedings of the Ottawa Linux Symposium, Berkeley, CA, USA, 2001. USENIX Association.

185. US National Institute of Standards and Technology: National vulnerability database. `http://nvd.nist.gov/`. [Accessed May 15, 2013].

186. Oliveira, F., Tjang, A., Bianchini, R., Martin, R. P., and Nguyen, T. D.: Barricade: defending systems against operator mistakes. In Proceedings of the 5th EuroSys Conference, pages 83–96, New York, NY, USA, 2010. ACM.

187. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In Proceedings of the 23rd Symposium on Operating System Principles, pages 159–172, New York, NY, USA, 2011. ACM.

188. Waterman, S.: Glitch imperils swath of encrypted records. The Washington Times, December 2012. http://www.washingtontimes.com/news/2012/dec/25/glitch-imperils-swath-of-encrypted-records/ [Accessed May 15, 2013].

189. Roscoe, T., Elphinstone, K., and Heiser, G.: Hype and virtue. In Proceedings of the 11th Workshop on Hot Topics in Operating Systems, pages 4:1–4:6, Berkeley, CA, USA, 2007. USENIX Association.

190. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A.: Xen and the art of virtualization. In Proceedings of the 19th Symposium on Operating System Principles, pages 164–177, Bolton Landing, NY, USA, October 2003. ACM.

191. Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W.: Diagnosing performance overheads in the Xen virtual machine environment. In Proceedings of the 1st ACM/USENIX International Conference On Virtual Execution Environments, pages 13–23, New York, NY, USA, June 2005. ACM.

192. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R.: An empirical study of operating system errors. In Proceedings of the 18th Symposium on Operating System Principles, pages 73–88, New York, NY, USA, 2001. ACM.

193. Wall, L., Christiansen, T., and Orwant, J.: Programming Perl. O'Reilly Media, 2000.

194. Rubio-González, C., Gunawi, H. S., Liblit, B., Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C.: Error propagation analysis for file systems. In Proceedings of the 2009 ACM Conference on Programming Language Design and Implementation, pages 270–280, New York, NY, USA, 2009. ACM.

195. Ford, B., Maren, K. V., Lepreau, J., Clawson, S., Robinson, B., and Turner, J.: The flux OS toolkit: Reusable components for OS implementation. In Proceedings of the 6th Workshop on Hot Topics in Operating Systems, pages 14–19, 1997.

196. Harper, J.: Re: Questions about attacks on Xen. xen-devel mailing list, January 2012. http://lists.xen.org/archives/html/xen-devel/2012-01/msg00916.html (accessed May 15, 2013).

197. Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K.: Overshadow: A virtualization-

based approach to retrofitting protection in commodity operating systems. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, WA, USA, March 2008.

198. Perez, R., van Doorn, L., and Sailer, R.: Virtualization and hardware-based security. IEEE Security and Privacy, 6(5):24–31, September 2008.

199. Ristenpart, T. and Yilek, S.: When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In Proc. of the Symp. on Network and Distributed Systems Security (NDSS). The Internet Society, 2010.

200. Murray, D. G., Milos, G., and Hand, S.: Improving Xen security through disaggregation. In Proceedings of the 4th ACM/USENIX International Conference On Virtual Execution Environments, pages 151–160, New York, NY, USA, 2008. ACM.

201. Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A.: Breaking up is hard to do: Security and functionality in a commodity hypervisor. In Proceedings of the 23rd Symposium on Operating System Principles, New York, NY, USA, October 2011. ACM.

202. Steinberg, U. and Kauer, B.: Nova: a microhypervisor-based secure virtualization architecture. In Proceedings of the 5th EuroSys Conference, pages 209–222, New York, NY, USA, 2010. ACM.

203. Dahlin, M., Johnson, R., Krug, R., McCoyd, M., Ray, S., and Young, B.: Toward the verification of a simple hypervisor. In 10th International Workshop on the ACL2 Theorem Prover and its Applications, November 2011.

204. Project, T. G.: Camel—a mail access library. `http://live.gnome.org/Evolution/Camel` [Accessed May 15, 2013].

205. Fisher, K. and Gruber, R.: PADS: a domain-specific language for processing ad hoc data. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 295–304, New York, NY, USA, 2005. ACM.

206. Bernstein, D. J.: Cryptography in NaCl. `http://cr.yp.to/highspeed/naclcrypto-20090310.pdf` (accessed May 15, 2013), 2009.

207. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S.: sel4: formal verification of an operating-system kernel. Communications of the ACM, 53(6):107–115, June 2010.

INDEX

W. Michael Petullo

Ph.D. Candidate

Education

2005 Master of Science in Computer Science, DePaul University (graduated with distinction)

1999 Bachelor of Science in Computer Science, Drake University

Publications

Petullo, W. Michael and Jon A. Solworth. "Digital identity security architecture in Ethos". In: *Proceedings of the 7th ACM workshop on Digital identity management*. DIM '11. (45% acceptance rate). Chicago, Illinois, USA: ACM, 2011, pp. 23–30.

— *Rethinking Operating System Interfaces to Support Robust Applications*. Poster Session of the 2012 IEEE Symposium on Security and Privacy. 2012.

— "Simple-to-use, Secure-by-design Networking in Ethos". In: *Proceedings of the Sixth European Workshop on System Security*. EUROSEC '13. (30% acceptance rate). Prague, Czech Republic: ACM, 2013.

— "Simple-to-use, Secure-by-design Networking in Ethos". Presentation at the 3rd ACM workshop on Runtime Environments, Systems, Layering and Virtualized Environments. Houston, Texas, USA, Mar. 2013.

— *The Ethos Project: Security Through Simplification*. Poster Session of the 2012 USENIX Symposium on Operating Systems Design and Implementation. 2012.

— "The Lazy Kernel Hacker and Application Programmer". Presentation at the 3rd ACM workshop on Runtime Environments, Systems, Layering and Virtualized Environments. Houston, Texas, USA, Mar. 2013.

Invited Talks

Petullo, W. Michael. *Let's Help Johnny Write Robust Applications*. Invited talk, December 3, University of Wisconsin–Madison. 2012.

Experience

1999-Present Currently a Major in the US Army. Most recently served as the Signal Center Director for 3rd Battalion, 3rd Special Forces Group; responsible for the planning and installation of all IP networks (LAN and TDMA/VSAT/WAN), IT systems, and combat network radio base stations (single-channel tactical satellite and HF) in support of a 300-man Special Operations Task Force conducting combat operations in Regional Command-East, Afghanistan.

Professional Activities

2005-Present	Member of ϒΠΕ, the International Honor Society for the Computing and Information Disciplines.
2011-2012	Student coordinator for the UIC Advanced Programming Seminar, a weekly seminar on applied topics related to programming (average attendance was 35 students).
2010	Google Summer of Code student mentor; my student added DACP support to libdmapsharing.
2009	Invited to help advise the United States Military Academy team during the NSA-sponsored Cyber Defense Exercise. Our winning team was featured in the New York Times.

Software Development

Flyn Computing is a project I started in 1999 to provide custom software solutions based on free software. Located at `http://www.flyn.org`, Flyn Computing hosts a collection of software I have developed and released as open source. I have provided services to companies such as Red Hat, Inc. and Cisco Systems, Inc.

2008-Present	Maintain libdmapsharing and dmapd, open source tools that interact with Apple's iTunes and iPhoto media sharing protocols.
2003-Present	Serve as a Fedora Project contributor; presently I maintain six packages.
1999-Present	Submitted nearly 60 source patches to a wide range of open source projects.
2008	Added support for encrypted boot partitions to GRUB.
2005	Implemented support for removable, encrypted disks for Fedora and Red Hat Enterprise Linux.
2004	Wrote pam-keyring, which was eventually integrated into GNOME's gnome-keyring.