Technical Report

UCAM-CL-TR-818
ISSN 1476-2986

Number 818

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

New approaches to operating system security extensibility

Robert N. M. Watson

April 2012

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2012 Robert N. M. Watson

This technical report is based on a dissertation submitted October 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

This dissertation proposes new approaches to commodity computer operating system (OS) access control extensibility that address historic problems with concurrency and technology transfer. Access control extensibility addresses a lack of consensus on operating system policy model at a time when security requirements are in flux: OS vendors, anti-virus companies, firewall manufacturers, smart phone developers, and application writers require new tools to express policies tailored to their needs. By proposing principled approaches to access control extensibility, this work allows OS security to be "designed in" yet remain flexible in the face of diverse and changing requirements.

I begin by analysing system call interposition, a popular extension technology used in security research and products, and reveal fundamental and readily exploited concurrency vulnerabilities. Motivated by these failures, I propose two security extension models: the TrustedBSD Mandatory Access Control (MAC) Framework, a flexible kernel access control extension framework for the FreeBSD kernel, and Capsicum, practical capabilities for UNIX.

The MAC Framework, a research project I began before starting my PhD, allows policy modules to dynamically extend the kernel access control policy. The framework allows policies to integrate tightly with kernel synchronisation, avoiding race conditions inherent to system call interposition, as well as offering reduced development and technology transfer costs for new security policies. Over two chapters, I explore the framework itself, and its transfer to and use in several products: the open source FreeBSD operating system, nCircle's enforcement appliances, and Apple's Mac OS X and iOS operating systems.

Capsicum is a new application-centric capability security model extending POSIX. Capsicum targets application writers rather than system designers, reflecting a trend towards security-aware applications such as Google's Chromium web browser, that map distributed security policies into often inadequate local primitives. I compare Capsicum with other sandboxing techniques, demonstrating improved performance, programmability, and security.

This dissertation makes original contributions to challenging research problems in security and operating system design. Portions of this research have already had a significant impact on industry practice.

Acknowledgements

Writing this dissertation would not have been possible without the support and encouragement of my family (especially my parents), friends, mentors, and colleagues, to whom I offer my sincerest thanks and appreciation.

Ross Anderson, my supervisor, deserves a special note of thanks: he has been supportive throughout my less than typical path through Cambridge's PhD programme, giving me space to pursue a variety of interests, many related to my PhD research, while still shepherding the dissertation to a timely completion! The security research group at the Computer Laboratory has been a critical source of experience and collaboration – especially Jonathan Anderson, Steven Murdoch, and Richard Clayton. Jon's work on Capsicum deserves particular note – the project could not have been completed without his help.

Early portions of my research into kernel access control extensibility, including the MAC Framework, were supported by DARPA and the US Navy via research contracts N66001-01-C-8035 (CBOSS[1]) and N66001-04-C-6019 (SEFOS[2]). This research took place at first TIS, then NAI Labs, later McAfee Research, and finally SPARTA ISSO. My colleagues at those institutions played central roles in helping me to take the idea of an extensible access control framework from concept, to prototype, to widely used system; many of them are acknowledged on page 11. However, I wish especially to thank Russ Mundy and Sandra Murphy, who mentored me through their own research projects in the TIS NetSec group. Special thanks are also due to Lee Badger, who at NAI helped me to formulate my first DARPA proposal around the concept of an extensible access control framework, and later at DARPA sponsored my seedling research project that became the inspiration for Capsicum.

Industrial collaboration has also played a central role in my PhD research: Google, Apple, nCircle Network Security, and Intel have engaged in elements of my research into access control extensibility and technology transfer. Google's university research grant programme supported my PhD research; Ben Laurie has not only promoted my research at Google, but also been an active collaborator in Capsicum. My thanks also go to Simon Cooper and Richard Gaushell for reviewing drafts of this dissertation.

Members of the global computer security research community have provided feedback on my research, including the anonymous reviewers of my papers. Most notably, Peter Neumann has provided both encouragement and detailed feedback on this dissertation. Comments and suggestions from my examiners, Jon Crowcroft (Cambridge) and Mark Handley (UCL), have made the dissertation much stronger, and are greatly appreciated.

I have drawn heavily on the open source community, whose products are remarkable assets for research that I have been proud to contribute to. The FreeBSD Project

[1]Community-Based Open Source Security

[2]Security Extensibility and Flexibility in Operating Systems

has provided seemingly endless assistance; all its contributors deserve my thanks, but especially John Baldwin, Pawel Dawidek, Poul-Henning Kamp, Kris Kennaway, Sam Leffler, George Neville-Neil, Colin Percival, and Bjoern Zeeb for their willingness to support and engage with my research over the last decade.

Dedication

I dedicate this dissertation to my wife and partner, Leigh Denault – I only hope that I supported you during your PhD as well as you have supported me!

Contents

1 Introduction **13**

1.1 Context for this research . . . 14
1.2 What is an operating system? . . . 15
1.3 Principles of operating system security . . . 17
1.3.1 Kernel and processes . . . 18
1.3.2 From isolation to access control policy . . . 19
1.3.3 Virtualisation . . . 21
1.3.4 Trusted systems . . . 22
1.3.5 Capability systems . . . 23
1.3.6 Of microkernels and security kernels . . . 24
1.3.7 Language and runtime approaches . . . 26
1.3.8 Extensible access control frameworks . . . 28
1.4 Structure of this dissertation . . . 30

2 Concurrency vulnerabilities in system call interposition **33**

2.1 Operating system kernels and concurrency . . . 34
2.2 Wrappers for security . . . 34
2.3 Concurrency attacks on wrappers . . . 35
2.4 Exploit techniques . . . 36
2.4.1 Concurrency approaches . . . 37
2.4.2 Racing on uniprocessor systems . . . 37
2.4.3 Racing on multiprocessor systems . . . 38
2.5 Exercising real vulnerabilities . . . 38
2.5.1 Generic Software Wrapper Toolkit (GSWTK) . . . 38
2.5.2 Systrace . . . 39
2.5.3 CerbNG . . . 41
2.6 Preventing wrapper races? . . . 42
2.6.1 Mitigation techniques . . . 42
2.6.2 Message passing systems . . . 43
2.6.3 Integrating security and concurrency . . . 43
2.7 Impact of the WOOT07 paper . . . 44
2.8 Conclusion . . . 45

3 The MAC Framework: extensible kernel access control 47
3.1 History of the MAC Framework 49
3.2 Past approaches 52
3.2.1 Direct modification 52
3.2.2 System call interposition 52
3.2.3 Stacked file systems 53
3.3 Limitations of past approaches 53
3.3.1 Kernel source code access 53
3.3.2 Tracking vendor development 54
3.3.3 Concurrency and lock order in threaded kernels 54
3.3.4 Policy composition 55
3.3.5 Financial cost of implementation 56
3.4 Designing for access control extension 57
3.4.1 Guiding principles 58
3.5 Architecture of the MAC Framework 60
3.5.1 Framework startup 61
3.5.2 Policy registration 62
3.5.3 Entry point design considerations 63
3.5.4 Kernel service entry point invocation 65
3.5.5 Policy entry point invocation 66
3.5.6 Policy composition 68
3.5.7 Object labelling 69
3.5.8 Application-layer approach 74
3.5.9 Policy-agnostic label management APIs 74
3.6 MAC Framework policy modules 75
3.6.1 The Biba integrity policy 75
3.7 Performance evaluation 79
3.7.1 System call performance 80
3.7.2 Network performance 86
3.7.3 Kernel build performance 88
3.8 Related work 89
3.9 Conclusion 89

4 The MAC Framework: from research to product 91
4.1 FreeBSD operating system 92
4.1.1 Experimental feature status 93
4.1.2 Performance 94
4.1.3 Third-party contributions to the MAC Framework 98
4.1.4 Additional MAC Framework consumers 100
4.2 nCircle IP360 monitoring appliance 101
4.2.1 What are system privileges? 102

4.2.2 System privilege extensions to the MAC Framework 102
4.2.3 The nCircle MAC policy 105
4.3 Apple's Mac OS X and iOS 106
4.3.1 SEDarwin research prototype 107
4.3.2 Adapting the MAC Framework to Mac OS X 108
4.3.3 Adoption by Apple 111
4.3.4 The Sandbox access control policy 112
4.3.5 Applications constrained by Sandbox 114
4.3.6 Enforcement in Mach and BSD 116
4.3.7 Paths in policy expression 116
4.3.8 Considerations for iOS 117
4.3.9 Performance optimisations 117
4.3.10 Policy label data synchronisation requirements 118
4.3.11 Conclusions on Mac OS X and iOS 119
4.4 Evaluation 120
4.4.1 The hypothesis of security extensibility 120
4.4.2 Expressiveness 122
4.4.3 Complexity 124
4.4.4 Usability 125
4.4.5 Performance 126
4.4.6 Security 126
4.5 Conclusion 128

5 Capsicum: practical capabilities for UNIX **131**
5.1 Introduction 132
5.2 Capsicum design 134
5.2.1 Capability mode 135
5.2.2 Capabilities 135
5.2.3 Run-time environment 139
5.3 Capsicum implementation 139
5.3.1 Kernel changes 139
5.3.2 The Capsicum run-time environment 140
5.3.3 Concurrency concerns with directory delegation 141
5.4 Adapting applications to use Capsicum 143
5.4.1 tcpdump 145
5.4.2 dhclient 146
5.4.3 gzip 148
5.4.4 Chromium 150
5.5 Comparison of sandboxing technologies 150
5.5.1 Windows ACLs and SIDs 150
5.5.2 Linux chroot 151

5.5.3 Mac OS X Sandbox 152
5.5.4 SELinux 152
5.5.5 Linux seccomp 153
5.5.6 Summary of Chromium isolation models 153
5.6 Performance evaluation 154
5.6.1 System call performance 154
5.6.2 Sandbox creation 155
5.6.3 gzip performance 155
5.7 Future work 159
5.8 Related work 159
5.9 Conclusion 160

6 Conclusions 163
6.1 Principles 164
6.1.1 Access control extensibility is a policy 164
6.1.2 Rehabilitating capabilities 165
6.1.3 The risks of software interposition 165
6.1.4 Technology transfer is research 166
6.1.5 A hybrid design philosophy 166
6.1.6 Open source infrastructure tech transfer 166
6.2 Future work 167
6.2.1 System call wrappers 167
6.2.2 The MAC Framework 167
6.2.3 Capsicum 168
6.2.4 CRASH-worthy Trustworthy Systems R&D (CTSRD) 169

Published work

In the course of this research, I have published the following research papers and articles that contributed directly to the contents of this dissertation:

The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0, published in the Proceedings of the 2003 USENIX Annual Technical Conference [145]. This article was co-authored with Wayne Morrison, Chris Vance, and Brian Feldman.

Design and Implementation of the TrustedBSD MAC Framework published by IEEE in the Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX III), 2003 [143]. This article was co-authored with Brian Feldman, Adam Migus, and Chris Vance.

Exploiting concurrency vulnerabilities in system call wrappers published in the proceedings of the first USENIX Workshop On Offensive Technologies (WOOT), 2007 [140].

Capsicum: practical capabilities for UNIX, published in the proceedings of the 19th USENIX Security Symposium, 2010 (best student paper) [142]. This article was co-authored with Jonathan Anderson, Ben Laurie, and Kris Kennaway.

During my dissertation research, I published a number of additional papers and articles in the areas of operating system and security research and development:

Building Systems to be Shared, Securely, published in ACM Queue 2004 [66]. This article was co-authored with Poul-Henning Kamp.

Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack, published in the Proceedings of EuroBSDCon 2005 [139].

The FreeBSD Audit System, published in the Proceedings of the 2006 UKUUG Spring Conference [144]. This article was co-authored with Wayne Salamon.

How the FreeBSD Project Works, published in the Proceedings of EuroBSDCon 2008 [141].

Ignoring the Great Firewall of China, first published in the Proceedings of the 2006 Privacy Enhancing Technologies Workshop [25]. Republished as an article in the Journal of Law and Policy for the Information Society 2007. This article was co-authored with Richard Clayton and Steven Murdoch.

Metrics for security and performance in low-latency anonymity systems, published in the Proceedings of the 2008 Privacy Enhancing Technologies Symposium [88]. This article was co-authored with Steven Murdoch.

Recipient of the BCS *Brendan Murphy MSN Young Researcher's Award* for the research *Network stack scalability and its protocol impact* presented at the Multi-Services Networks 2010 workshop.

The Age of Avatar Realism: When Seeing Shouldn't be Believing, published in IEEE Robotics and Automation Magazine, 2010 [108]. This article was co-authored with Laurel Riek.

Work done in collaboration

As with all significant systems research, portions of the work described in this dissertation were performed in collaboration with others, whom I would like to acknowledge:

The MAC Framework

The TrustedBSD MAC Framework, which I designed while a research scientist at NAI Labs and have continued to work on during my PhD, was implemented with the support of the NAI Labs research staff, as well as industrial, academic, and open source collaborators. My collaborators on this project have included Lee Badger, Samy Bahra, John Baldwin, Chris Costello, Simon Cooper, Hrishikesh Dandekar, Rob Dekelbaum, Brian Feldman, Mark Feldman, Tim Fraser, Shawn Geddis, Ilmar Habibulin, Mike Halderman, Doug Kilpatrick, Suresh Krishnaswamy, Terry Lambert, Pete Loscocco, Jim Magee, Adam Migus, Todd Miller, Wayne Morrison, Christian Peron, Andrew Reisse, Tom Rhodes, Wayne Salamon, Mike Smith, Tom Van Vleck, Chris Vance, and Zhouyi Zhou.

Capsicum

I have been supported in the implementation and evaluation of Capsicum by colleagues at the University of Cambridge and Google UK Ltd. My collaborators on this project were Jonathan Anderson, Ben Laurie, and Kris Kennaway.

Chapter 1

Introduction

This dissertation addresses a critical research question: how can computer operating system (OS) security be "built-in from the ground up" yet remain flexible in the face of changing requirements? The last fifteen years have demonstrated that this question is a central concern in OS design, as commodity systems have been thrust from isolation into near-ubiquitous and remarkably malicious networking environments. This transformation has been catalysed by fundamental changes in computer hardware: decreased size, increased availability, and dramatic performance improvements, allowing operating systems originally designed for network servers to be used in notebook computers, DSL routers, and smart phones.

Operating systems are the foundation on which software systems are built, and a firm security foundation is central to the security of the applications that run above it. This firm foundation has become more and more necessary: countless previously "dumb" devices now not only run non-trivial software stacks, but are also continuously exposed to malicious environments. Yet there has been a surprising lack of consensus about system security policy, motivating an interest in systems that support a variety of access control policies yet commit firmly to none. Growing from several years of work on system security in this context, this dissertation argues that designing for security extensibility allows operating systems to be adapted to new requirements without requiring them to be rewritten from scratch.

This dissertation stems from my work in operating system security as a DARPA principal investigator at NAI Labs and its successor organisations. I have brought insights from new research conducted while at the University of Cambridge Computer Laboratory that have transformed the original projects, tracking the adoption and evolution of new ideas about security extensibility.

The theme of extensibility is the central concern of this dissertation. It has also been a key concern in operating system research over a long period, although the motivations for, and definitions of, extensibility have developed considerably over time. The work described in this dissertation focuses on commodity operating systems: ones that are widely used and have strong security requirements, yet have not taken the

high assurance approaches described later in this chapter. Transformations in computer usage have motivated fresh investigation and renewed application of ideas from past operating system and security literature: concurrency, discretionary and mandatory access control, reference monitors, and capabilities.

We begin with a consideration of the current and historic context in which new ideas about security have developed, an introduction to key ideas in operating system security, and a detailed chapter outline.

1.1 Context for this research

My DARPA-sponsored research from the early 2000s focused on operating system security models derived from the needs of mandatory access control environments in which a system-wide access control policy is dominant. This interest spanned traditional information flow policies such as Biba [18] and Bell-La Padula (BLP) [15], discussed in detail later in this chapter, but also hardening policies that would later be integrated into products such as smart phones, routers, and firewalls. I have revisited that research by analysing its motivations (especially with respect to concurrency and integrated access control), implementation (and reimplementation), and its impact on industry.

Invaluable experience with the TrustedBSD MAC Framework (described in Chapter 3) in deployment was gained through collaboration with academic, industrial, and government collaborators, including a close collaboration with Apple during their development of Mac OS X and iOS security features, Seccuris in their development of monitoring appliances, and nCircle in development of a policy enforcement appliance. Additional feedback and contributions from MAC Framework consumers such as SCC (later McAfee) and Juniper have also been critical, as the hypothesis of an extensible kernel security model matured. For these vendors, security extensibility has offered immediate and concrete benefits: improved security, a principled model on which security extensions can be built, and long-term reduction in cost when maintaining local security features. This experience confirms the hypotheses of Doug Maughan's DARPA Composable High Assurance Trusted Systems (CHATS) research programme, which funded portions of my initial work on the MAC Framework: not only was there the opportunity to improve the state of the art through research into security and composability, but also that open source can serve as a effective means of technology transfer.

Shortly before my move to the Computer Laboratory, I worked on a seedling research project sponsored by Lee Badger, then DARPA programme manager, entitled Visibly Controllable Computing. My analysis revealed an application-centric view of the world, and illustrated the need to integrate security functionality into the structure of increasingly complex (and monolithic) applications that, themselves, had clear but unfulfilled security requirements. Among applications considered were desktop managers, web browsers, and office suites which act on behalf of a user in interacting with

documents from various mutually untrusting (and perhaps even malicious) origins, but require, at any given moment, few of the ambient rights of their users. This analysis is central to the capability security model espoused in Capsicum.

Mapping distributed security properties into local enforcement, as explored in Capsicum, is a particularly interesting aspect of this problem. The narrow single-system views of OS vendors rarely serve the needs of the authors of complex distributed systems that will be based on them. All of these prior research projects have demonstrated that we still do not fully understand what security policies will best meet our needs. Security extensibility is, fundamentally, about planning infrastructure for poorly understood requirements, and while necessarily an imperfect approach, it is the only one that enables us to build flexibility into our systems to allow for unanticipated future developments. Direct support for security extensibility, rather than adopting the access control model of the moment, therefore offers a constructive way to address this fundamental problem in operating system design.

1.2 What is an operating system?

The history of operating systems is long and contentious. As such, this review of the evolution and concepts of operating systems and security skims some topics and periods, aiming at the problems that most concern this dissertation. Contemporary OSs have their grounding in 1960s and 1970s time sharing systems, and retain many of the principles developed in that period:

- abstraction of system hardware,
- resource management: accounting, scheduling, and synchronisation,
- storage and communication services: file systems, network stack, inter-process communication (IPC),
- an application model, typically premised on process separation,
- libraries of common functions: mathematics, compression, etc., and
- management of user interaction and interface.

Throughout the 1960s to 1990s, new functionality evolved in mainframes, microcomputers, servers, and high end workstations: hardware support for robustness (especially memory protection), security, and networking, as well as graphical user interfaces, multitasking, multiprocessing, and virtualisation. In contrast, personal computers of the 1980s and early 1990s were single address-space, weakly connected to other computers (if at all), and the primary form of software distribution was via floppy disk ("sneaker net"). By the end of the decade, higher-end technologies had slid downmarket to the personal workstations, notebook computers, tablets, and smart phones of the 2000s.

Fundamental new technologies emerged in the consumer space as well, including digital subscriber lines (xDSL), local- and wide-area wireless networking, making personal computing devices the epicentre, rather than periphery, of computer security.

As a result, requirements for security have ballooned to include features previously found only in research or high assurance "trusted" systems, as well as new technologies necessary to address the world of distributed systems unanticipated by earlier systems and security research. Some of these features centre around Anderson's concept of a trusted computing base (TCB) – the self-protecting core in the operating system providing confidence in its security [5]. Others place individual systems securely in a global network context through services built on cryptography and cryptographic protocols, also central security research products of the 1980s and 1990s. Commonly available systems now provide, to varying degrees, the following types of security features:

- a TCB based on stronger assumptions of isolation and access control than a simple process model,
- support for trusted/verified boot,
- security co-processors responsible for key management and cryptography,
- authentication and multiplexing of multiple simultaneous users,
- discretionary and mandatory access control models,
- security event auditing for accountability,
- cryptography for data integrity and confidentiality,
- code authentication via digital signatures,
- distributed security models (e.g., Kerberos, x.509 certificates, and TLS),
- cryptographically protected network communications (e.g., SSH, IPsec),
- sandboxing facilities to contain potentially malicious mobile code, and
- secure update to remedy vulnerabilities discovered after manufacture.

These features are found in products ranging from smart phone operating systems such as Symbian, iOS, and Android, to workstation and server operating systems such as Windows, Mac OS X, FreeBSD, Linux, and Solaris.

1.3 Principles of operating system security

As with many aspects of contemporary computer and operating system design, the origins of operating system security may be found in the Massachusetts Institute of Technology's (MIT) Project MAC. The project began with MIT's Compatible Time Sharing System (CTSS) [27], and continued over the next decade with MIT's Multics project, and would develop many central tenets of computer security [28, 55]. Dennis and Van Horn's 1965 *Programming Semantics for Multiprogrammed Computations* [33] for the first time laid out principled hardware and software approaches to concurrency, object naming, and security for multi-programmed computer systems – or, as they are known today, multi-tasking and multi-user computer systems. Multics implemented a coherent, unified architecture for processes, virtual memory, and protection, integrating new ideas such as *capabilities*, unforgeable tokens of authority, and *principals*, the end users with whom authentication takes place and to whom resources are accounted [112].

In 1975, Saltzer and Schroeder surveyed the rapidly expanding vocabulary of computer security in *The Protection of Information in Computer Systems* [113]. They enumerated design principles such as the *principle of least privilege*, which demands that computations run only with the privileges they require, as well as the core security goals of protecting *confidentiality*, *integrity*, and *availability*. The tension between fault tolerance and security, a recurring debate in systems literature, saw its initial analysis in Lampson's 1974 *Redundancy and Robustness in Memory Protection* [75], which considered how hardware memory protection addressed both types of failure.

The security research community also blossomed outside of MIT: Wulf's Hydra operating system at Carnegie Mellon University (CMU) [152, 26], Needham and Wilkes' CAP Computer at Cambridge [146], SRI's Provably Secure Operating System design (PSOS) [40, 96], Rushby's security kernels supported by formal methods at Newcastle [111], and Lampson's work on formal models of security protection at the Berkeley Computer Corporation all explored the structure of operating system access control, and especially the application of capabilities to the protection problem [73, 74]. Another critical offshoot from the Multics project was Ritchie and Thompson's UNIX operating system at Bell Labs, which simplified concepts from Multics, becoming the basis for countless directly and indirectly derived products such as today's Solaris, FreeBSD, Mac OS X, and Linux operating systems [109].

The creation of secure software went hand in hand with analysis of security flaws: Anderson's 1972 US Air Force *Computer Security Technology Planning Study* not only defined new security structures, such as the *reference monitor* described in Section 1.3.8, but also analysed potential attack methodologies such as Trojan horses and inference attacks [5]. Karger and Schell's 1974 report on a security analysis of the Multics system similarly demonstrated a variety of attacks bypassing hardware and OS protection [69]. In 1978, Bisbey and Hollingworth's *Protection Analysis: Project final report* at ISI identified common patterns of security vulnerability in operating system design, such

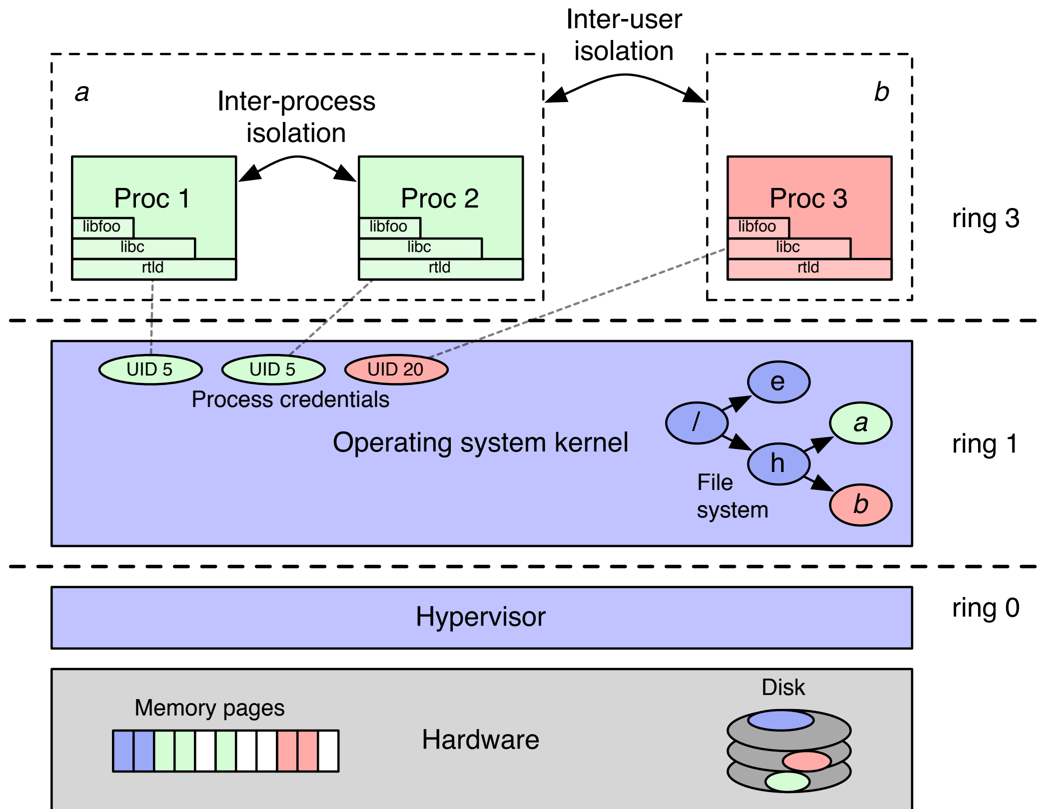

Figure 1.1: A sketch of the contemporary OS structure and security model: the OS administers hardware resources, such as memory pages and disk blocks, to provide higher-level services such as processes, virtual memory, and file storage. CPU rings provide a hierarchical basis for isolation; on this foundation, credentials bound indelibly to user processes allow access control policies to be constructed. Numbering of rings is not consistent across architectures; other rings may also exist for firmware services such System Management Mode (SMM).

as race conditions and incorrectly validated arguments at security boundaries [19]. Adversarial analysis of system security remains as critical to the success of security research as principled engineering and formal methods.

Forty-five years of research have explored these and other concepts in great detail, bringing new contributions in hardware, software, language design, and formal methods, as well as networking and cryptography technologies that transform the context of operating system security. However, the themes identified in that period remain topical and highly influential, structuring current thinking about systems design.

1.3.1 Kernel and processes

Most current operating systems employ a derivative of the Multics process model, illustrated in Figure 1.1. In the prototypical model, described by Saltzer, hardware features separate the OS kernel from user applications, and applications from one another, offering improved robustness [112]. Programs execute in isolated *processes*, each a virtual machine with its own address space, able to initiate communication directly with only

the OS kernel. If multiple instances of a programme are desired, then multiple processes are used, providing independence of failure modes. The process model relies on two CPU features: *rings*, which limit use of privileged chip functions to a kernel operating in *supervisor mode*, and virtual addressing, which allows the kernel to control the mapping of user process pointers into physical addresses. Additional hardware rings, described by Schroeder, provide hierarchical protection, a feature largely exploited today for the purposes of virtualisation, as discussed in Section 1.3.3 [118].

The operating system kernel executes in a protected address space, managing memory, servicing inter-process communication (IPC), and scheduling processes. In most operating systems, it also hosts central system services: device drivers, file systems, and networking – the trend towards *microkernel* systems, in which those services are themselves hosted in processes, is described in Section 1.3.6. User processes invoke operating system kernel services by initiating hardware traps, including system calls (explicit) and virtual memory faults (implicit). The hardware arranges a transition between rings and virtual memory contexts as the kernel regains control; kernel trap handlers must be carefully crafted to prevent accidental leaks of information or control between it and the user process, or between user processes.[1]

Whereas the Multics model called for further granular hardware controls of memory, such as fine-grained segmentation and capabilities as found in Cambridge's CAP Computer [146], current hardware generally provides only coarse controls at a page granularity. Karger has explored the performance implications of various hardware designs for security domain transition [68], and Wahbe's software fault isolation, described later in this chapter, has also considered the performance impact of this practical design choice [134]. Witchel has more recently explored alternative x86 memory protection models in Mondrix, derived from the Multics design, and tested it with a modified Linux kernel under simulation, although this work was targeted at the kernel rather than applications [147]. However, the end result is that current hardware designs prevent the easy subdivision of applications into separate security domains – a limitation that can be overcome using message-passing, at significant cost to performance and ease of representation, or through the use of type-safe language runtimes and virtual machines, at a significant cost in complexity. These approaches are considered in detail in Sections 1.3.6 and 1.3.7.

1.3.2 From isolation to access control policy

The process model provides robustness by limiting application communication with the kernel to controlled paths, and by preventing errant memory access in one application from corrupting the memory of another. This separation is also the foundation for security isolation, as the kernel can impose policies over whether and how exceptions

[1]This interface has historically been a source of frailty – recent `NULL` pointer vulnerabilities in Linux have allowed accidental (and harmful) kernel access to user addresses [128].

to strict isolation are permitted[2]. For example, most operating systems mediate access to inter-process communication (IPC), file system storage, and the network stack.

Kernels indelibly tag processes with *credentials* that hold security properties used for access and resource control – the integrity of credentials is protected by placing them in the kernel address space[3]. Commonly, credentials hold the user ID of the authenticated principal on whose behalf a process is acting, additional groups the user is a member of, and mandatory access control state, all of which are checked against access control policies.

Discretionary access control (DAC) allows users to administer protections on objects they own, and is frequently specified using Access Control Lists (ACLs), a technique pioneered in the Multics file system [29]. This approach, most commonly used in file systems, assigns sets of rights (such as read, write, and execute/lookup) to lists of principals associated with each object. The utility of DAC is clear in multi-user systems, where it can be used to structure collaboration, but DAC's role in single-user computers such as desktops, tablets, and mobile phones is less obvious. At least one system, the Android phone OS, assigns a separate UNIX user to each application, allowing DAC to control interactions between applications [52]. However, historic criticisms of DAC – that it is hard to manage, and worse, that its discretionary and intricate nature leads easily to configuration errors resulting in security problems – are relevant in both multi-user and mobile phone settings [7].

In contrast, mandatory access control systemically enforces policies representing the interests of system implementers and administrators. Information flow policies tag process credentials and objects with sensitivity and integrity labels, enforcing fixed rules to prevent reads or writes that could lead to information leakage. Multi-Level Security (MLS), formalised as Bell-LaPadula (BLP), protects confidential information from unauthorised release and is modelled on the military classification model [15]. MLS's logical dual, the Biba integrity policy [18], implements a similar scheme protecting integrity, and is sometimes used to protect TCBs, such as in Argus's Pitbull product [12]. Fraser's Low-Watermark Mandatory Access Control (LOMAC) puts a dynamic spin on the Biba integrity policy by allowing subject labels to float, tracking taint on processes to protect system integrity [45].

Early high assurance systems, such as PSOS, founded security policy definition and enforcement on capabilities; however, Boebert has raised concerns regarding the ability of pure capability systems to implement mandatory policies such as MLS[4] [21]. PSOS's

[2]While some operations persist beyond the end of a system call, such as setting up shared memory, they must be initiated using an explicit system call that can be controlled.

[3]There have been notable cases in which this has not been done properly: historic UNIX systems stored credentials in the *U-area* mapped directly above user process memory – incorrect protections in one UNIX release allowed user processes to overwrite their credential!

[4]Kain and Landwehr have argued that this limitation applies only to specific capability system semantics, and discuss two alternative models in which the *-property is maintained [64]. Miller, *et al.*, provide a more impassioned (and perhaps more accessible) defence of capabilities in *Capability*

strong enforcement of object types evolved into Boebert's type enforcement (TE) [22], employed in NSA and Honeywell's PSOS-derived LoCK operating system [114]; Badger has extended TE in domain and type enforcement (DTE) and applied it to lower-assurance UNIX designs [13]. In TE and DTE, system subjects are labelled with *domains*, and objects are labeled with *types*, whose interactions are controlled at every access by a configurable rule set.

TE's flexibility allows it to support many applications, including *assured pipelines*, in which a series of processes are linked to perform stepwise processing between two endpoints, such as two network interfaces in a firewall: data can flow between the two endpoints only via the steps in the pipeline. TE has, as a result, seen extensive use in both dedicated-purpose systems such as SCC's high assurance Sidewinder Firewall (discussed further in Chapter 3) [79], but also in the widely used SELinux [78], and has been experimentally deployed for FreeBSD and Mac OS X [131]. In SELinux, TE's flexibility is also its weakness, requiring complex policy files that have proven difficult to write and maintain, a concern I explore further in Chapter 5. I consider the microkernel context in which TE was developed in more detail in Section 1.3.6.

Another pertinent direction in access control policy was the Compartmented Mode Workstation (CMW), a computer system design intended for environments handling classified intelligence data in which data of varying levels and compartments had to be processed by users. Notable research prototype CMW systems were developed at Mitre [16] and TRW [37]; later production CMW systems include Trusted Solaris [129]. CMW systems provided the context for applying mandatory access control to windowing systems, incorporating ideas about data labelling and access control to the user interface. These ideas have been influential in the contemporary world of web browsers and Java applets [51], as well as virtualisation in research systems such as Nitpicker [41] and Xen3D [126].

1.3.3 Virtualisation

Operating system virtualisation is another potential application of the hierarchical hardware rings and protection-oriented virtual memory used to implement process models. This approach inserts a *hypervisor* in a ring above the hardware and below the operating system. This model has been implemented on mainframes, such as IBM's CP-67, since the 1970s [132].

Additional rings (long-present in the Intel architecture) have seen heavy use in full operating system virtualisation at the lower end of the market with VMWare since the 1990s, which overcame non-virtualisable instructions on Intel systems through dynamic code rewriting [133]. More recently, open source systems such as Xen have also become widespread [14], and the Intel instruction set has been improved to more easily support virtualisation[5]

Myths Demolished [154].

[5]The ring and virtualization story on Intel is further complicated by the fact that virtual machine

The primary thrust of virtualisation has been making efficient use of increasingly capable (and hence idle) hardware by allowing independent services to be collocated. However, the isolation properties of virtualisation have also been of interest in the security arena, as suggested by Anderson in 1972 [5]. For example, NSA's NetTop platform relies on a blend of virtualisation of low-assurance guest operating systems and a higher-assurance host with mandatory access control to provide strong separation [92].

1.3.4 Trusted systems

In 1983, the US Department of Defense's (DoD) National Computer Security Center (NCSC) released the Trusted Computer System Evaluation Criteria (TCSEC), or Orange Book [91]. This document became the benchmark by which a new class of *trusted computer systems* were designed, developed and evaluated, and consisted primarily of guidance on the security features and assurance properties required for military computing systems.

Access control features such as DAC and MAC policies are prescribed, as well as accountability through detailed logging of security-relevant events, sometimes referred to as a *secure audit trail* or simply *security event auditing*. Beyond security features, TCSEC required *assurance*, or a structured argument supported by evidence that the design, implementation, and operation of a system is trustworthy with respect to security.

In the Orange Book, assurance is divided into two categories: *life-cycle assurance*, referring to the design, implementation, testing, and maintenance of the system in order to ensure correctness, and *operational assurance*, the utilisation of hardware and software techniques to support secure behaviour. At high levels of assurance, techniques such as mathematical proof are applied to design and then implementation, made easier by formalisms such as the TCB[6]. Likewise, hardware-supported techniques, such as multiple security domains (commonly implemented using rings and memory protection), provide operational guarantees that are more easily reasoned about.

The requirement that computer systems used by DoD must be evaluated under TCSEC motivated a generation of computer system vendors to collaborate with NCSC to produce successively more feature-complete and higher-assurance systems. However, a series of deferrals of this deadline permitted continued use of unevaluated systems in DoD well into the 2000s.

The TCSEC was replaced during the late 1990s by the ISO-standardised Common Criteria (CC) evaluation process – in the CC model, evaluations are with respect to Protection Profiles (PPs) identifying security requirements for specific environments[7] [97].

---

monitors may run in "root mode", which might logically be thought of as ring -1.

[6]In many Orange Book systems, the TCB consisted not only of a UNIX kernel, but also all of the userspace components necessary to bootstrap user login – a sizeable portion of UNIX libraries and commands!

[7]The Common Criteria also replaced other national standards, hence "common".

Use of Common Criteria evaluation is now widespread, with most general-purpose commercial operating system products evaluated against one or more OS-centric protection profiles as a matter of course. Frequently used protection profiles include the Common Access Protection Profile (CAPP) [58] and Labeled Security Protection Profile (LSPP) [59], that mandate, respectively, discretionary and mandatory access controls.

Despite considerable criticism of the Orange Book and Common Criteria, often grounded in their social and economic contexts, as well as debate over the methodology of formal evaluation processes, it is clear that these standards have had significant impact on the design and implementation of operating systems [6]. Not least, they have motivated the addition of features such as security event auditing that would otherwise be omitted from commercial, off-the-shelf products such as Microsoft Windows and Mac OS X[8].

1.3.5 Capability systems

Over the next few sections, we consider three closely related ideas: capability security, microkernel OS design, and language-based constraints. These apparently disparate areas of research are linked by a duality, first observed by Morris in 1973, between the enforcement of data types and safety goals in programming languages and the hardware and software protection techniques explored in operating systems [87]. Each of these approaches blends a combination of limits defined by static analysis (perhaps at compile-time), limits on expression on the execution substrate (such as what programming constructs can even be represented), and dynamically enforced policy that generates run-time exceptions (often driven by the need for configurable policy and labelling not known until the moment of access). Different systems make different uses of these techniques, affecting expressibility, performance, and assurance.

Dennis and Van Horn's security design was influential throughout the 1970s and 1980s [33], providing a model for the integration of software and hardware security adopted by both research and industrial computing systems. Capabilities, of particular interest in this dissertation, saw investigation in academia through a variety of hardware prototypes considering various semantics for protection. These included Ackerman's modifications to the DEC PDP-1 architecture at MIT [4], Wilkes and Needham's CAP Computer at Cambridge [146], and Cohen and Levin's Hydra computer at CMU [26, 76].

Each of these systems selected various points along several spectra of implementation choices, leading to valuable research results. Hydra develops a number of ideas, including the relationship between capabilities and object references, developing the *object-capability* paradigm, as well as pursuing the separation of policy and mechanism[9].

---

[8] In 2003, I led the team that developed OpenBSM, the audit framework used in FreeBSD and Mac OS X, and believe that audit has significant value regardless of evaluation, for historic accountability reasons, but also as a data source for intrusion detection and security analysis systems [144].

[9] Miller has expanded on the object-capability philosophy in considerable depth in his 2006 PhD dissertation, *Robust composition: towards a unified approach to access control and concurrency con-*

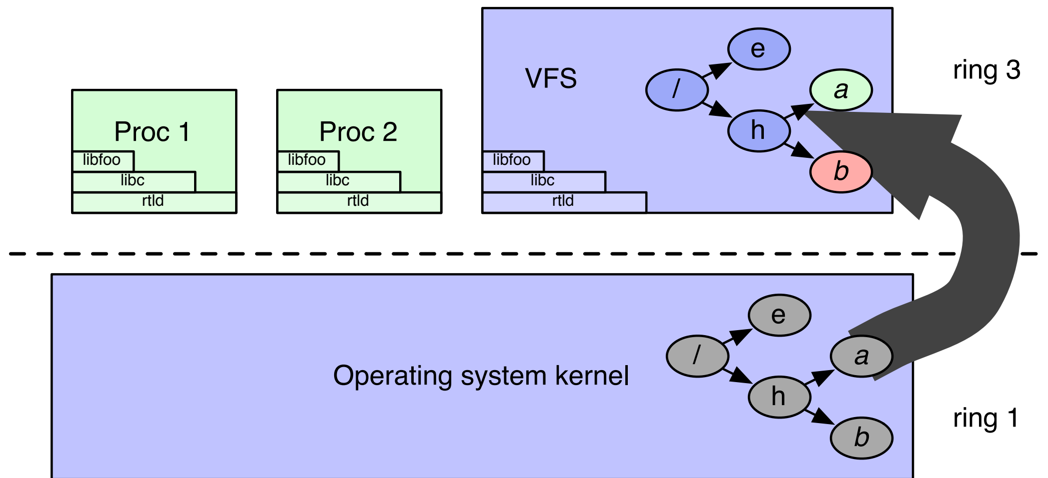

Figure 1.2: The microkernel project shifts complex OS components, such as file systems, from the kernel to userspace tasks linked by IPC. Microkernels provide a smaller, easier to analyse, easier to debug, and more robust foundation in the face of dramatic increases in OS complexity.

Jones and Wulf argue through the Hydra design that the capability model allows the representation of a broad range of system policies as a result of integration with the OS object model, which in turn facilitates interposition as a means of imposing policies on object access, an approach discussed in Section 1.3.8 [62]. The design tension between expressibility and access control became more clear: whereas some systems (such as Hydra) do not permit the expression of invalid accesses since capabilities cannot be constructed to represent them, the CAP computer imposes capability evaluation in the memory access path, leaving the instruction set largely conventional and trapping to specialised microcode on illegal references.

1.3.6 Of microkernels and security kernels

Denning has argued that the failures of capability hardware projects were classic failures of large systems projects, an underestimation of the complexity and cost of reworking an entire system design, rather than fundamental failures of the capability model [32]. However, the benefit of hindsight suggests that the demise of hardware capability systems is a result of three related developments in systems research: microkernel OS design, a related interest from the security research community in security kernel design, and Patterson and Sequin's Reduced Instruction Set Computers (RISC) [100].

Successors to Hydra at CMU include Accent and Mach [104, 3], both microkernel systems intended to explore a decomposition of a large and decidedly un-robust operating system kernel. Figure 1.2 illustrates the principle of microkernel design: traditional OS services, such as the file system, are migrated out of ring 0 and into user processes, improving debuggability and independence of failure modes. They are also based on

trol [84]

mapping of capabilities as object references into IPC pipes (*ports*), in which messages on ports represent methods on objects.

This shift in operating system design went hand-in-hand with a related analysis in the security community: Lampson's model for capability security was, in fact, based on pure message passing between isolated processes [74]. This further aligned with proposals by Andrews [8] and Rushby [111] for a *security kernel*, whose responsibility lies solely in maintaining isolation, rather than the provision of higher-level services such as file systems. Unfortunately, the shift to message passing also invalidated Fabry's semantic argument for capability systems: that by offering a single namespace shared by all protection domains, the distributed system programming problem had been avoided [38].

A panel at the 1974 National Computer Conference and Exposition (AFIPS) chaired by Lipner brought the design goals and choices for microkernels and security kernels clearly into focus: microkernel developers sought to provide flexible platforms for OS research with an eye towards protection, while security kernel developers aimed for a high assurance platform for separation, supported by hardware, software, and formal methods [77].

The notion that the microkernel, rather than the hardware, is responsible for implementing the protection semantics of capabilities also aligned well with the simultaneous research (and successful technology transfer) of RISC designs, which eschewed microcode by shifting complexity to the compiler and operating system. Without microcode, the complex C-list peregrinations of CAP's capability unit, and protection domain transitions found in other capability-based systems, become less feasible in hardware. Simple virtual memory designs based on fixed-size pages and few semantic constraints have since been standardised throughout the industry.

Security kernel designs, which combine a minimal kernel focused entirely on correctly implementing protection, and rigorous application of formal methods, formed the foundation for several secure OS projects during the 1970s. Schiller's security kernel for the PDP-11/45 [116] and Neumann's Provably Secure Operating System [96] design study were ground-up operating system designs grounded in formal methodology[10]. In contrast, Schroeder's MLS kernel design for Multics [117], the DoD Kernelized Secure Operating System (KSOS) [44], and Bruce Walker's UCLA UNIX Security Kernel [135] attempted to slide MLS kernels underneath existing Multics and UNIX system designs. Steve Walker's 1980 survey of the state of the art in trusted operating systems provides a summary of the goals and designs of these high assurance security kernel designs [136].

The advent of CMU's Mach microkernel triggered a wave of new research into security kernels. TIS's Trusted Mach (TMach) project extended Mach to include mandatory access control, relying on enforcement in the microkernel and a small number of security-related servers to implement the TCB, accomplishing sufficient assurance to

[10]PSOS's ground-up design included ground-up hardware, whereas Schiller's design revised only the software stack.

target a TCSEC B3 evaluation [23]. Secure Computing Corporation (SCC) and the National Security Agency (NSA) adapted PSOS's type enforcement from LoCK for use in a new Distributed Trusted Mach (DTMach) prototype, building on the TMach approach while adding new flexibility [120]. DTMach, adopting ideas from Hydra, separates mechanism (in the microkernel) from policy (implemented in a userspace security server) via a new reference monitor framework, FLASK. [127] A significant focus of the FLASK work was performance: an access vector cache is responsible for caching access control decisions throughout the OS in order to avoid costly up-calls and message passing (with associated context switches) to the security server. NSA and SCC eventually migrated FLASK to the FLUX microkernel developed by the University of Utah in the search for improved performance. This flurry of operating system security research, invigorated by the rise of microkernels and their congruence with security kernels, also faced the limitations (and eventual rejection) of the microkernel approach by the computer industry, which perceived the performance overheads as too great.

Microkernels and mandatory access control have seen another experimental composition in the form of Decentralized Information Flow Control (DIFC). This model, proposed by Myers, allows applications to assign information flow labels to OS-provided objects, such as communication channels, which are propagated and enforced by a blend of static analysis and runtime OS enforcement, implementing policies such as taint tracking [90] – effectively, a composition of mandatory access control and capabilities in service to application security. This approach is embodied by Efstathopoulos et al's Abestos [36] and Zeldovich et al's Histar [156] research operating systems.

Despite the decline of both hardware-oriented and microkernel capability system design, capability models continue to interest both research and industry. Shapiro's EROS (now CapROS) continues the investigation of higher-assurance software capability designs [124], and was inspired by the proprietary KEYKOS system [56].

More influentially, Morris's suggestion of capabilities at the programming language level has seen widespread deployment. Gosling and Gong's Java security model blends language-level type safety with a capability-based virtual machine [54, 51]. Java maps language-level constructs (such as object member and method protections) into execution constraints enforced by a combination of a pre-execution bytecode verification and expression constraints in the bytecode itself. Java has seen extensive deployment in containing potentially (and actually) malicious code in the web browser environment. Miller's development of a capability-oriented E language [84], Wagner's Joe-E capability-safe subset of Java [83], and Miller's Caja capability-safe subset of JavaScript continue a language-level exploration of capability security [85].

1.3.7 Language and runtime approaches

Direct reliance on hardware for enforcement is central to both historic and current systems, but not the only approach to enforcing isolation. The notion that limits on expressibility in a programming language can be used to enforce security properties is

frequently deployed in contemporary systems in order to supplement coarse and high-overhead operating system process models. Two techniques are widely used: virtual machine instruction sets (or perhaps physical machine instruction subsets) with limited expressibility, and more expressive languages or instruction sets combined with typing systems and formal verification techniques.

The Berkeley Packet Filter (BPF) is one of the most frequently cited examples of the virtual machine approach: user processes upload pattern matching programs to the kernel, in order to avoid data copying and context switching when sniffing network packet data [80]. These programs are expressed in a limited packet filtering virtual machine instruction set capable of expressing common constructs, such as accumulators, conditional forward jumps, comparisons, etc., but incapable of expressing arbitrary pointer arithmetic that could allow escape from confinement, or control structures such as loops that might lead to unbounded execution time. Similar approaches have been used via the type safe Modula 3 programming language in SPIN [17], and the DTrace instrumentation tool which, like BPF, uses a narrow virtual instruction set to implement the D language [24].

Google's Native Client (NaCl) model edges towards a verification-oriented approach, in which programs must be implemented using a "safe" (and easily verified) subset of the x86 or ARM instruction sets, allowing confinement properties to be validated [153]. NaCl relates closely to Software Fault Isolation (SFI) [134], in which safety properties of machine code are enforced through instrumentation to ensure no unsafe access, and Proof-Carrying Code (PCC) in which the safe properties of code are demonstrated through attached and easily verifiable proofs [94]. As mentioned in the previous section, the Java Virtual Machine (JVM) model is similar, combining run-time execution constraints of a restricted, capability-oriented bytecode with a static verifier run over Java classes before they can be loaded into the execution environment, ensuring that only safe accesses have been expressed. C subsets, such as Cyclone [60], and type-safe languages such as Ruby [110], offer similar safety guarantees, which can be leveraged to provide security confinement of potentially malicious code without hardware support.

These techniques offer a variety of trade-offs relative to CPU enforcement of the process model: some (BPF, D) limit expressibility that may prevent potentially useful constructs from being used, such as loops bounded by invariants rather than instruction limits, as well as imposing a potentially significant performance overhead. Systems such as FreeBSD often support just-in-time compilers (JITs) that convert less efficient virtual machine bytecode into native code subject to similar constraints, addressing performance but not expressibility concerns [82].

Systems like PCC that rely on proof techniques have had limited impact in industry, and often align poorly with widely deployed programming languages (such as C) that make formal reasoning difficult. Type-safe languages have gained significant ground over the last decade, with widespread use of JavaScript and increasing use of functional languages such as OCaML [107], and offer many of the performance benefits

with improved expressibility, yet have had little impact on operating system implementations. However, an interesting twist on this view is described by Wong in Gazelle, in which the observation is made that a web browser is effectively an operating system by virtue of hosting significant applications and enforcing confinement between them [137]. Web browsers frequently incorporate many of these techniques including Java Virtual Machines and a JavaScript interpreter.

1.3.8 Extensible access control frameworks

Policy flexibility and extensibility have been a consideration in operating system design since the inception of operating system access control. Multics supported two forms of access control extensibility: first, *trap* entries in access control lists, allowing customised policy code to execute, and second, general use of a capability security architecture, allowing delegation of capabilities to encapsulate application-specified policies. These two related principles have been explored in considerable depth, and derivatives of both have appeared in many systems. A third principle is the separation between policy and mechanism, valuable in assurance arguments, which enables extensibility as policy becomes increasingly pluggable, if cleanly separated from mechanism. Anderson *et al.*'s 1972 *Computer Security Technology Planning Study* proposes that a distinct *reference monitor* implement policy, with the following guarantees:

- The reference validation mechanism must be tamper proof.
- The reference validation mechanism must always be invoked.
- The reference validation mechanism must be small enough to be subject to analysis and tests to assure that it is correct.

The method by which a reference monitor is integrated with the operating system, however, is left open.

Levin *et al.* argue not only for a principled separation of policy and mechanism in Hydra, but also for a practical implementation in which the kernel provides the foundations for a secure object system, enforcing but not determining policy [76]. Jones and Wulf provide an elegant philosophical argument for an extensible approach in which the same protection mechanisms provided for the operating system should be extended to all processes running on the system [62]. In Hydra, that mechanism is an object capability model describing both OS objects and objects maintained by processes, in which capabilities incorporate not just references to underlying objects, but also a list of named rights that a given capability permits on the object. Because the semantics of these accesses (or in current parlance, *methods*) does not need to be visible to the operating system, certain policies can be enforced blind to those semantics, permitting the object system to be extensible without change being required to the mechanisms of enforcement. Jones and Wulf further observe that a variety of useful constructs can be created using object interposition on capabilities, including revocation (also observed

by Redell [105]) and filtering. Karger demonstrates the flexibility and power of the interposition approach as an alternative to mandatory access control in containing trojan horses [67].

Systems such as Hydra, TMach, DTMach, and DTOS achieve, to varying degrees, the goal of separating policy from mechanism and providing strong enforcement via reference monitors; however, those systems have failed to see widespread adoption due to their reliance on microkernels. FLASK itself represents a form of access control extensibility, as a generalisation of type enforcement through an extensible interface [127].

Throughout the 1990s, access control researchers also explored the application of various security policies to medium- and low-assurance designs, including trusted UNIX systems integrating MLS, fine-grained privileges, and occasionally, integrity protection, as well as experimentation with alternative security policies in systems such as Linux and FreeBSD. However, OS vendors found themselves in the awkward position of having to support various access control models without a clear vision for what would become a widely used feature verses an expensive-to-maintain local extension.

System call interposition, discussed in detail in Chapter 2, is an operating system extension technique premised on "wrapping" system calls from within the kernel address space in order to impose new semantics – not dissimilar to capability interposition techniques. This approach was widely used in access control research in the 1990s, and persists in research and commercial products despite strong evidence (in part raised as a result of research in this dissertation) that the approach is fundamentally flawed in concurrent systems. The approach is appealing for security product vendors as it does not require modifying, or even access to, the operating system's source code, and offers easy access to the system call interface. A related approach relies on the use of the `ptrace` debugging system call to introduce new constraints on users processes without modifying the kernel, and frequently suffers from similar vulnerabilities.

Abrams' Generalized Framework for Access Control (GFAC) suggests one way out: adoption of the idea of a reference monitor, but unlike high assurance or microkernel approaches, within the confines of the existing operating system kernel [2]. Ott implemented the GFAC model for Linux in Rule Set Based Access Control (RSBAC), and influenced future open source OS developers to consider this direction [99]. Both the TrustedBSD MAC Framework (2000), discussed in Chapters 3 and 4 [143], and shortly thereafter, Linux Security Modules (LSM) (2001) [148, 39] are grounded in access control uncertainty: the desire for a reference monitor without a commitment to a single policy. Resulting extensibility permitted the adaption of policies as varied as Biba, MLS, LOMAC, type enforcement, and more UNIX-centric models within these frameworks.

Apple has also explored pluggable access control in its Mac OS X and iOS operating systems, used respectively in Mac computers and "embedded" iPhone, iPod Touch, and iPad platforms. Initially, Apple took the approach of developing a new extension framework, KAuth, or Kernel Authorisation Framework, to satisfy the needs of third-

party developers such as anti-virus vendors on Mac OS X, an approach also adopted by the NetBSD Project [9]. However, in later releases, the TrustedBSD MAC Framework was added to Mac OS X and iOS to support Apple's own more comprehensive security models, such as the Sandbox sandboxing, Quarantine data tainting framework, and integrity protection in Apple's Time Machine backup system. Sandbox is used to constrain potentially risky network services on Mac OS X, and enforce isolation in the iPhone [1]. These policies are considered in greater detail in Chapter 4.

1.4 Structure of this dissertation

This dissertation describes new analyses and technologies that contribute to understanding and implementing operating system access control extensibility. Validation of these approaches through research prototypes is a critical element of the dissertation, and so each chapter not only explores theory, but also the implications of implementation and methods of validation.

Chapter 1 has introduced central OS security constructs: isolation, the principle of least privilege, discretionary and mandatory access control models, and capabilities. I also considered the importance of changing hardware and security requirements, which motivated much of this research.

Chapter 2 considers system call interposition, a security extensibility technology popular through the 1990s and 2000s in operating system security research and products (such as anti-virus software). Analysis of the approach reveals significant and readily exploited concurrency vulnerabilities.

Chapter 3 investigates a new approach to operating system security extension: a flexible access control framework. The TrustedBSD MAC Framework is tightly integrated with the kernel concurrency model, allowing third-party policy modules to avoid inherent race conditions.

Chapter 4 picks up where Chapter 3 leaves off, exploring the successful technology transfer of the MAC Framework through a series of case studies: the FreeBSD operating system, nCircle's enforcement appliance, and Apple's Mac OS X and iOS operating systems. I consider the diverse set of requirements placed on the framework across several operating systems, and the enhancements made to the framework from its original design, especially relating to SMP performance and policy expressiveness, that have led it to be a success.

In Chapter 5, I turn my attention to OS support for application security extensibility through Capsicum, a new OS security model that blends historic UNIX design with a capability model. Capsicum focuses on application writers rather than system designers, reflecting a trend from multi-user computers to single-user systems with security-aware applications. Concurrency plays a key design role, and we explore one concurrency vulnerability that lends itself to formal analysis through model checking.

This dissertation makes novel contributions in each of these areas, bringing new

solutions to historically challenging research problems in operating system security. I conclude in Chapter 6, deriving a set of principles for the imposition and structure of access control in operating systems, gleaned from practical experience. These may be of import to future system designs, as well as shedding light beyond the narrow field in computer security.

Chapter 2

Concurrency vulnerabilities in system call interposition

To motivate new work in operating system security extension, it is necessary to understand why prior prior approaches in the commodity OS space are inadequate. This chapter explores *system call interposition*, a kernel extension technique used extensively to augment operating system security policies without modifying underlying OS code. System call wrappers are widely used in research systems and commercial anti-virus software despite research suggesting security and reliability problems; Garfinkel [48], Ghormley [50], and the author [143] have all previously described the potential for concurrency vulnerabilities in wrapper systems.

This chapter is based on a paper presented at the First USENIX Workshop On Offensive Technologies (WOOT07), which was intended to discredit system call wrappers as an approach to access control extension – a practice that persisted due to a belief that proposed concurreny vulnerabilities were purely theoretical. The paper provided detailed and hands-on consideration of concurrency attacks on system call interposition, and has been cited as a key consideration in design choices for later security research. Building on that work, I investigate vulnerabilities and exploit techniques for real-world systems, and demonstrate that inherent concurrency problems lead directly to exploitable vulnerabilities, concluding that addressing these systemic vulnerabilities requires rethinking security extension architecture.

This chapter first introduces concurrency in operating system kernels and the system call wrapper technique. I then discuss the structure of concurrency vulnerabilities, the applicability of concurrency attacks to wrappers, and practical exploit techniques. Next, I investigate privilege escalation and audit bypass vulnerabilities in three system call interposition systems, Generic Software Wrappers Toolkit (GSWTK) [46], Systrace [102], and CerbNG [30]. Finally, I explore deployed mitigation techniques and architectural solutions to these vulnerabilities. As I examine new approaches to extensibility in later chapters, I return not just to these specific attacks, but to more general concerns with concurrency and security that arise in security research.

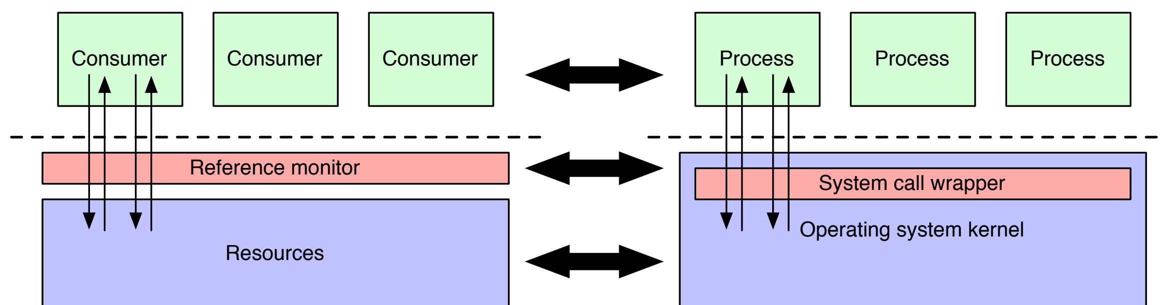

Figure 2.1: Misleading congruence of reference monitor and system call wrappers: despite appearances, the request path intercepted by wrappers is bypassable.

2.1 Operating system kernels and concurrency

Concurrency is a fundamental operating system feature, and must be considered carefully in any security design. Operating system kernels, themselves, are highly concurrent programs, producing and consuming concurrency services internally, as well as offering them to applications. Concurrency in operating system kernels arises from two hardware sources: interrupts resulting from timers, network events, etc, and parallelism deriving from hardware multiprocessing. Most desktop and server systems support multiprocessing in some form, as do increasing numbers of embedded systems (such as smart phones), traditional bastions of minimalism.

Concurrency also arises in operating system kernels as a result of the asynchronous and event-oriented nature of internal kernel services and those offered to applications running on the OS. Kernels provide internal threading facilities to kernel subsystems (file systems, network stacks, etc) and expose concurrency to applications via processes, threading, signals, and asynchronous input and output (I/O). Kernel subsystem authors and application writers employ these facilities to mask I/O latency, exploit hardware parallelism, and structure programs serving multiple consumers.

2.2 Wrappers for security

System call interposition imposes a *reference monitor* on kernel services by intercepting system calls (Figure 2.1). As described in Chapter 1, Anderson states that a reference monitor must be tamper proof, always invoked, and small enough to be subject to analysis and tests to assure correctness [5]. On face value, system call wrappers appear to meet these criteria: they run in the kernel's protection domain, are invoked in the system call path, and avoid complex modifications to kernel source code by isolating security decisions in a self-contained module.

Further, wrappers work at the level of the UNIX application programming interface (API), allowing many wrapper packages to be portable across multiple operating systems. System call wrappers are compiled into the kernel or loaded as a module,

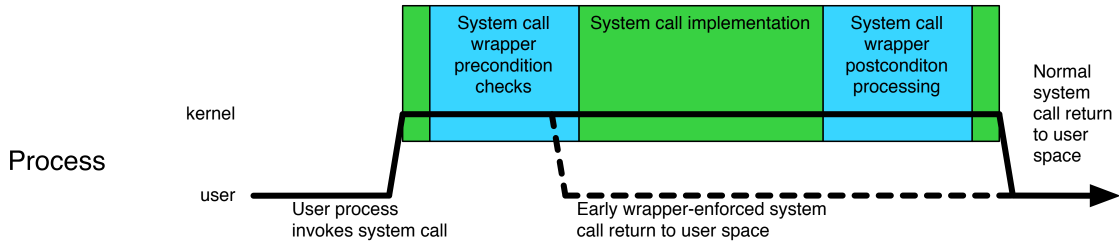

Figure 2.2: System call wrappers implement precondition and postcondition processing around the system call.

hooking the system call trap handler (Figure 2.2). I adopt terminology from GSWTK:

- *Precondition* hooks execute before the system call runs, and inspect or substitute arguments before kernel services see them.
- *Postcondition* hooks execute prior to returning control to user space, allowing them to track and log results, transform return values, etc.

Wrappers perform policy checks with the system call number and arguments, and have access to kernel data structures such as process control blocks. On access control failure, they may abort the call, transform arguments, modify credentials, log events, or cause other side effects. By substituting argument and return values, system call wrappers can change system object name spaces visible to applications; e.g., a wrapper can redirect file open requests or change the IP address bound by an application. Policy sources include compiled policies (LOMAC) [45], policy configuration languages (GSWTK, CerbNG), and even message passing to user processes (Systrace).

2.3 Concurrency attacks on wrappers

As highly concurrent software protecting critical data, operating system kernels are fertile ground for the discovery of concurrency vulnerabilities. In contrast to assumptions of atomic system call made in previous considerations of race conditions, key to our approach is non-atomicity between the kernel and system call wrappers [31]. I have identified and successfully exploited three forms of concurrency vulnerability:

- Synchronisation bugs in wrapper logic leading to incorrect operation, e.g., improper locking of data.
- Lack of synchronisation between the wrapper and the kernel in *copying* system call arguments, such that argument values processed by the wrapper and the kernel differ. I describe these as *syntactic* race conditions.

- Lack of synchronisation between the wrapper and the kernel in *interpreting* system call arguments, as kernel and wrapper interpretations of identical argument values differ. I describe these as *semantic* race conditions.

The latter two forms involve not a wrapper in isolation, but rather its failure in composition with the service it protects.

I focus on syntactic race conditions that do not depend on kernel and wrapper internals, and hence are portable across wrapper frameworks and operating systems. I found that the most frequently identifiable and exploitable vulnerabilities fall into three categories:

- *Time-of-check-to-time-of-use* (TOCTTOU) vulnerabilities, in which access control checks are non-atomic with the operations they protect, allowing an attacker to violate access control rules.
- *Time-of-audit-to-time-of-use* (TOATTOU) vulnerabilities, in which the trail diverges from actual accesses as a result of non-atomicity, violating accuracy requirements. This allows an attacker to mask activity, avoiding IDS triggers.
- *Time-of-replacement-to-time-of-use* (TORTTOU) vulnerabilities[1], unique to wrappers, in which attackers modify system call arguments after a wrapper has replaced them but before the kernel has accessed them, violating the security policy.

I am not aware of prior research on audit and replacement vulnerabilities.

2.4 Exploit techniques

Concurrency vulnerabilities exist where there is inadequate synchronisation of a shared resource leading to violation of security policy. The first step in locating concurrency vulnerabilities is to identify resources relevant to access control, audit, or other security functionality that are accessed concurrently across a trust boundary. Relevant resources include file system objects, shared memory, and sockets, as well as indirectly accessed kernel objects, such as nodes/inodes and kernel buffers. To exploit race conditions in interposition, I use process memory, accessed by the user process, wrapper, and kernel, to hold system call arguments. User memory is accessed from the kernel with copying functions, e.g., BSD `copyin()` and `copyout()`, which validate addresses and page memory as needed.

Direct arguments are passed as stack variables or via registers, and contain values such as process IDs and pointers; they are copied in by the system call trap handler. Wrappers consume the same instance of these arguments as the kernel, so are not subject to syntactic races.

[1]Neumann has observed that this is a torturous acronym.

Indirect arguments are referenced by pointers, often passed as direct arguments, and copied on-demand by kernel services: for example, file paths are copied and resolved by `namei()`. Indirect arguments are copied after the precondition hook, so wrappers copy them independently from the kernel, opening a race window between the two copy operations. These races are limited to indirect arguments, many of which are security-critical, such as socket addresses, file paths, arguments to `ioctl()` and `sysctl()`, group ID lists, resource limits, and I/O data.

2.4.1 Concurrency approaches

Concurrent program execution on UNIX occurs via signals, asynchronous I/O, threads, and processes. Operations in a single process necessarily support shared memory; processes may share memory using `minherit()`, `rfork()`, and `clone()`, explicit shared memory and, debugging interfaces. For the purposes of the experimental exploits in this chapter, I share memory across processes by inheritance, as other methods are not consistent across systems. Prior work has considered races between user processes, but I am interested in races between user threads and the kernel itself. This requires the user thread and kernel to run concurrently, which is possible through interleaved scheduling or hardware parallelism.

2.4.2 Racing on uniprocessor systems

On uniprocessor (UP) systems, the attacker must cause the kernel to yield to a user thread between execution of the system call and wrapper preconditions and postconditions. Yielding may occur voluntarily (a thread requests blocking I/O on a socket or disk) or involuntarily (a kernel thread accesses memory resulting in a page fault from disk). Both may be used to race with system call wrapper preconditions and postconditions.

Page faults on indirect system call arguments are effective in opening up race windows. The resulting wait on disk I/O provides a scheduling window of several million instruction cycles, more than enough time for an attacking thread to replace the contents of a memory page. On most systems it is easy to arrange for user memory to be paged to disk, either swap (if configured) or a memory mapped file, by increasing memory pressure.

Initially, I believed that this technique was limited to system calls with multiple indirect arguments, such as `rename()`. I was able to successfully exploit this case by paging the `rename()` target path to disk, allowing the source path to be replaced between check and use. On reflection, I realised that copy operations themselves are non-atomic, sleeping part-way through if user data spans multiple pages, allowing even system calls with a single argument to be attacked. This works well as the last byte of many indirect arguments is not essential: strings are null-terminated, and many data structures contain padding. Page faults may also be used to attack postconditions, subject to the limitation that it is possible to force a page fault on each page only once

during most system calls.

Voluntary thread sleeping also prove useful: during a TCP `connect()`, the calling thread will wait in the kernel for a TCP ACK to confirm the connection, allowing a user thread to execute a postcondition attack on auditing after the arguments have been copied in by the kernel.

2.4.3 Racing on multiprocessor systems

For the purposes of this work, I consider any system with cache-coherent shared memory parallelism to be a multiprocessor (MP), including SMT and multicore systems. UP systems are vulnerable to races, but require manipulating kernel scheduling via a limited set of yield opportunities. On MP systems, races between user and kernel threads can be exploited without relying on kernel sleeping, as user threads may run simultaneously on other CPUs. Inter-CPU shared memory race conditions are narrow, as they occur at memory speed (10K-100K cycles) rather than disk or network speed (>1M cycles), and the challenge remains timing memory modifications so that the wrapper sees one version of an argument while the kernel sees another.

I use two approaches to identify timing for argument replacement. In the case of argument copies without replacement, a binary search for the Time Stamp Counter (TSC) length of the race window can be performed by timing the results of the system call being raced with; as many of these races are effectively deterministic, successful exploitation is simply a matter of timing. In the case of argument copies with replacement by the wrapper, it is possible to simply spin while watching for the replacement to take place in-memory, and then modify the argument to its original value, or a new value.

I found that race window length varied based across wrapper systems. Races on arguments in GSWTK, which runs only in kernel, were often between 5K and 15K cycles. Systrace passes control to a user process, which performs optional copies in and out of the target process, opening race windows of over 100K cycles. The order of magnitude difference in race window size did not, however, lead to measurable differences in exploit effectiveness: I had a 100% success rate in exploiting races across packages.

2.5 Exercising real vulnerabilities

As the goal of this approach is to demonstrate the immediate practicality of exploiting race conditions in real-world interposition systems, I now consider hands-on experiments in doing so. All experiments and measurements were performed on a 3.2 GHz Intel Xeon.

2.5.1 Generic Software Wrapper Toolkit (GSWTK)

GSWTK is a kernel access control system that allows task-specific system call wrappers to inspect and modify arguments and return values. Wrappers are written using a

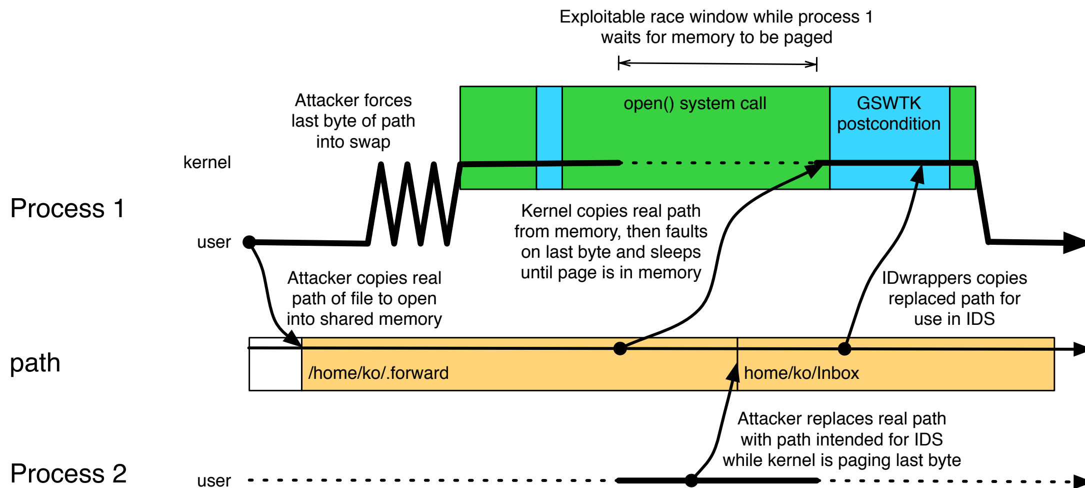

Figure 2.3: Processes employ paging to force `copyin()` in `open()` to sleep so that the process can use a TOATTOU attack on an intrusion detection wrapper.

C language extension with integrated SQL database support. GSWTK is available as a third-party package on the Solaris, FreeBSD, BSD/OS, and Linux platforms; I employed GSWTK 1.6.3 on FreeBSD 4.11. A variety of wrappers are available, from access control policies to intrusion detection systems.

I was able to successfully substitute values used in both precondition access control and postcondition auditing and intrusion detection on UP with paging, and on MP systems from a second processor. After experimentally validating the approach on a subset of wrappers, I inspected the remaining wrappers shipped with GSWTK. Of 23 wrappers available for UNIX or all platforms, 16 had one or more vulnerabilities (Table 2.1). Also of interest is Ko's work on sequence-based intrusion detection, as it illustrates the potential impact of TOATTOU vulnerabilities [72]. Investigation revealed vulnerabilities in several intrusion detection wrappers. Figure 2.3 illustrates one such race, in which an intrusion detection wrapper intended to detect an exploited `imapd` vulnerability is bypassed by racing between the kernel open, which sees the forbidden pathname `/home/ko/.forward`, and the post-condition wrapper that monitors it, which sees to permissible pathname `/home/ko/Inbox`.

2.5.2 Systrace

Systrace is an access control system that allows user processes to control target processes by inspecting and modifying system call arguments and return values. The OpenBSD operating system includes Systrace; NetBSD has done so historically, and there are ports available to Mac OS X, FreeBSD, and Linux. For this work, I used Systrace on NetBSD 3.1, 4.0 (Jan. 2007), and OpenBSD 4.0. As Systrace is a programmable policy system, I used two policies: Sudo monitor mode [86] and Sysjail [61]. I bypassed protections in both, violating access control policy and audit trail integrity.

Wrapper	Description	Vulnerabilities
`callcount`	Count system calls	None: uses system call number.
`conwatch`	Track IP connections by processes.	Postcondition TOATTOU race on `connect()` and `bind()` masks actual address/port.
`dbfencrypt`	Encrypt files with '$' in their names; prevent rename so that policy cannot be changed.	Postcondition TOCTTOU race allows incorrect name in policy check; precondition TORTTOU races on I/O write unencrypted data and bypass rename checks.
`dbexec`	Authorise program execution based on a pathname database.	Precondition TOCTTOU race allows bypass by substituting the name during the wrapper check.
`dbsynthetic`	Synthetic file system sandbox substituting pathnames.	Precondition TORTTOU race bypasses path replacement; postcondition TORTTOU race leaks true paths
`life`	Prints process life cycle.	Precondition TOATTOU race replaces `exec()` paths.
`noadmin`	Deny all privileged operations.	None: relies on the kernel's process credential.
`aks.wr`	Audit file operations	Pre/postcondition TOATTOU races avoid audit.
`seq-kernel.wr`	Sequence-based intrusion detection	None: uses system call number.
`imapd.wr`	Detect anomalous access by `imapd`.	Postcondition TOATTOU path races prevent alerts.

Table 2.1: Examples of concurrency vulnerabilities in GSWTK and ID Wrappers.

Sudo

Sudo is a widely used privilege management tool allowing users to run authorised commands with the rights of another user [86]. The prerelease version of Sudo includes a "monitor mode", implemented using Systrace, that audits commands executed by Sudo-derived processes. Sudo's systrace monitor intercepts invocations of `execve()` that occur after a successful user switch, auditing indirectly executions by the command. The `execve()` system call accepts a program path, command line arguments,

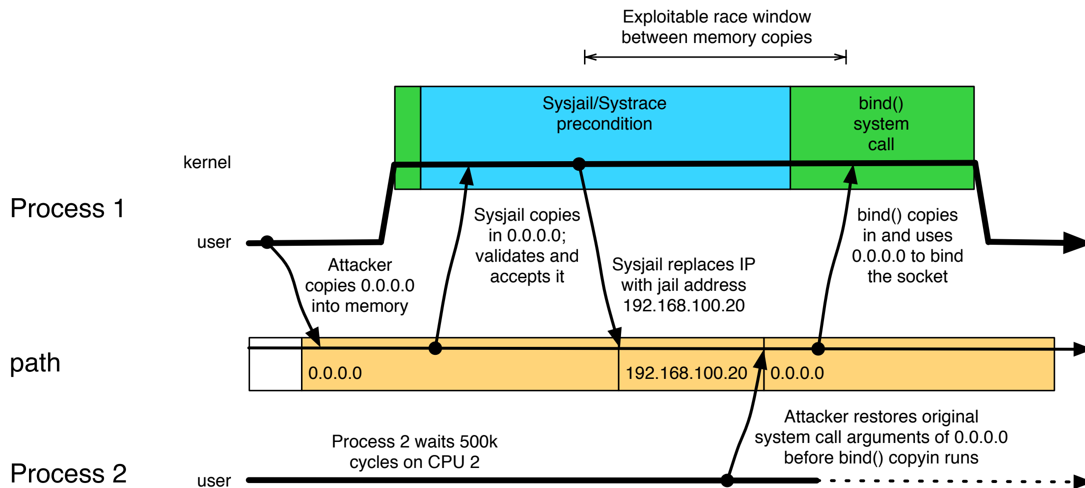

Figure 2.4: Race to bypass protections from a second processor by replacing the IP address passed to `bind()` between check and use.

and environmental variables as indirect arguments, and thus is vulnerable to attack. Due to a user-space policy source, Systrace requires additional context switches to make access control decisions, leading to larger race windows. With Sudo on MP systems, the window for `execve()` arguments was over 430K cycles. I was able to successfully exploit this vulnerability, replacing command lines so that they were incorrectly logged, masking all further attacker activity in the audit trail.

Sysjail

Sysjail is a port of the FreeBSD jail containment facility using the Systrace framework for NetBSD and OpenBSD [61, 65]. Sysjail attaches to all processes in the jail, validating and in some cases rewriting system call arguments to maintain confinement. Sysjail is of particular interest as it is intended to contain processes running with root privilege, increasing exposure in the event of vulnerability.

Sysjail handles several indirect arguments, including IP addresses passed to `bind()`. It enforces two constraints: the address must be configured for the jail or it must be `INADDR_ANY`, in which case it will be replaced with the jail's address. By racing with the Sysjail, I am able to replace the IP accepted by Sysjail with another IP address, bypassing network confinement (Figure 2.4).

2.5.3 CerbNG

CerbNG is a third-party security framework for FreeBSD 4.8 similar to GSWTK. It allows rule-based control of system calls, checking and modifying arguments and return values, changing process properties, and logging events. I successfully exploited TOATTOU and TOCTTOU races in rules shipped with the system, replacing command lines in `log-exec.cb`, which audits `execve()`, generating incorrect audit trails.

CerbNG unsuccessfully employs several virtual memory defences discussed in Section 2.6.1.

2.6 Preventing wrapper races?

System-call wrapper races can lead to partial or complete bypass of access control and audit. To address this, concurrency must be properly managed. I consider proposed solutions in three areas: those that retain the wrapper architecture but modify wrapper systems to mitigate attacks, those that retain the wrapper architecture but modify the OS kernel, and those that entirely abandon the wrapper approach in extending kernel security.

2.6.1 Mitigation techniques

Lee Badger (private communication) has suggested a weak consistency approach: detect and mitigate exploitative changes in kernel state via a postcondition, taking remedial action if a violation has occurred. I believe that this approach faces challenges from the complex side effects of some system calls (e.g., `connect()` and `unlink()`); detecting inconsistency faces the same atomicity issues as other postconditions.

Pawel Dawidek (private communication) has experimented with marking memory pages that hold system call arguments as read-only during system calls. If implemented properly, this prevents argument races, but violates concurrent programming assumptions. Legitimate multithreaded processes may store concurrently accessed data in the same memory page as arguments, and will suffer ill effects such as unexpected faults.

VM protection is nontrivial, as all mappings of a physical page must be protected. One interesting case involves memory-mapped files: systems with unified VM/buffer caches must prevent writes via I/O system calls, not just mapped memory. Protecting pages is also insufficient: the address space must be protected to prevent the unmapping of protected pages and remapping with writable ones. I found several vulnerabilities in CerbNG's VM protections, including incorrect write protection of pages, and race windows while copying arguments.

Provos provides similar facilities in Systrace, copying indirect arguments into the "stack gap", a reserved area of process memory, allowing wrappers to substitute indirect arguments of greater size than the original argument. He has also suggested that this may be used to resist shared memory attacks as the stack gap area is unique to each process. This protection is not effective with threads, as threads share a single address space. Experimentation on OpenBSD indicates that the stack gap mapping can be replaced with shared memory accessible to other processes even in the non-threaded case. This approach also causes additional data copies for any protected arguments, which for large arguments may significantly impact performance both by virtue of additional copying and memory footprint.

Ghormley addresses argument races in the SLIC interposition framework via in-kernel buffers that extend the user address space. Each thread caches regions of the address space copied by the extension or kernel; future accesses will be from the cache, preventing further modification by user threads. This approach requires replacing the kernel copy routines so that the kernel and the wrappers use the cache. As cache buffers are not forced to page size, the false sharing effects of page protections are avoided; however, this approach imposes a significant performance penalty, as all indirect arguments must be copied and cached in kernel memory.

VM and caching schemes make processing indirect arguments that are read and written in a single system call (such as POSIX asynchronous I/O) more tricky. None of the systems with protections were able to handle this case correctly, although this had limited impact as none of the sample policies controlled affected system calls.

Many of these mitigation techniques suffer serious correctness and performance problems. VM and caching protect only against syntactic vulnerability, as they prevent the attacker from replacing arguments and do not synchronise with kernel services. Fundamentally, system call wrappers are not architecturally well-placed to synchronise with the kernel, as this conflicts with clean separation from the kernel.

2.6.2 Message passing systems

In order to maintain the system-call interposition model without resorting to mitigation techniques, kernel operation must be changed. One possibility is to move to a message-passing model, in which system call arguments are bundled and delivered to the kernel at once rather than being copied on-demand. This approach would not eliminate semantic race conditions, but would eliminate syntactic race conditions by allowing wrappers to inspect the same argument values as the kernel. The disadvantage to this model is that it requires the complete layout of arguments to be available to the trap handler; currently, this knowledge is distributed across many layers of the kernel.

Garfinkel's Ostia [49] and Seaborn's Plash [119] both implement message-passing approaches in which access to the file system name space must occur "by proxy" via a monitor process, avoiding argument and name space copying races, but allowing further accesses to occur directly using a passed file descriptor – a technique also employed, with measurable overhead, in my own Capsicum design, discussed in Chapter 5. VM mitigation schemes may be gradually extended to approximate the message passing paradigm, although they provide less clean implementations than systems designed with message passing in mind, such as Mach [3].

2.6.3 Integrating security and concurrency

A more flexible, if more complex approach, is to eliminate race conditions between security extensions and the kernel by integrating security checks with the kernel itself. Invocations of security extensions then occur throughout the kernel, atomically with respect to use of the objects they control. For example, access control checks on a

process operation would be performed while holding locks on the process to prevent changes in associated context.

As system call interposition was developed to avoid OS modification, this may seem contradictory; however, the move to open source systems and the adoption of security extensions has driven the creation of security frameworks by vendors in attempts to maintain the ideals of a reference monitor. This approach has been adopted by FLASK in SELinux [127], SEBSD [131], and SEDarwin, the TrustedBSD MAC Framework in FreeBSD and Mac OS X [143], kauth in Mac OS X [9] and NetBSD [35], and Linux Security Modules [148]. The degree of integration varies across systems: at one extreme, the TrustedBSD MAC Framework, described in detail in Chapter 3, enforces object locking at each entry from the kernel, allowing policies to rely on kernel locks to protect associated access control checks. At the other extreme, the kauth framework allows up-calls to a user process, which precludes holding some locks over checks: this re-introduces the opportunity for races, but as with systrace, offers the benefit of userspace-originated policy.

Integrated kernel security frameworks do not eliminate the problem of concurrency vulnerabilities entirely, but they do make it possible to avoid race conditions innate to the system call interposition approach.

2.7 Impact of the WOOT07 paper

Since the publication of my WOOT07 paper, the practicality of the attacks it describes has influenced several OS vendors to select integrated security approaches rather than system call interposition, especially in the arena of smart phone security. Research has also demonstrated that similar vulnerabilities exist in large numbers of Windows security and anti-virus products. In 2010, commercial consulting firm matousec.com reported that 35 windows anti-virus systems they analysed could be bypassed using system call wrapper race conditions, illustrating the broad applicability of the approach [34].

Researchers have continued to explore alternative approaches to security extensibility – most notably, systems such as the MAC Framework, LSM, and file system stacking have become the preferred models for extensibility. For example, Ford and Cox's Vx32 sandboxing scheme specifically addresses concurrency attacks [43], as does Potter's PeaPod via stacked file system use [101]. Fetzer's SwitchBlade is limited to direct argument processing and temporal analysis in part because of the potential for concurrency issues [42]. Many citations to this paper now exist in security and systems research; unfortunately, new interposition-based research has continued to be published merely citing the potential for races, rather than addressing them [98].

2.8 Conclusion

In this chapter, I have explored concurrency vulnerabilities in system-call interposition security extensions, arguing that correctness with respect to concurrency is critical to access control and audit. I demonstrated that several wrapper systems suffer from common classes of concurrency vulnerabilities allowing privilege escalation and intrusion detection bypass.

These vulnerabilities derive from the fundamental architectural separation of the wrapper from native kernel synchronisation strategies – the same structural separation that leads to an appealing but deceptive similarity to an idealised reference monitor. I also demonstrated that many deployed mitigation solutions suffer from vulnerabilities, as well as semantic and performance degradations, and that architectural solutions require much tighter integration of security with the kernel.

While the problems described here are technical in nature, they are also structural. Perhaps more fundamentally, these structural problems reflect a lack of OS vendor involvement in security extensibility. In the next chapter, I consider approaches by which OS vendors can directly support the needs of security extension authors. Such facilities have also proven valuable to OS vendors themselves, as they attempt to adapt their operating system products for changes in security requirements.

Chapter 3

The MAC Framework: extensible kernel access control

This chapter describes the TrustedBSD MAC Framework, a kernel access control extension framework for the FreeBSD and Mac OS X operating systems. The MAC Framework formalises the relationship between kernel subsystems and the policy extensions that will control them. An explicit facility for pluggable access control policy extension brings a number of benefits:

1. Pluggable policy necessitates separation of policy from enforcement, as enforcement is scattered throughout the kernel. This introduces the structure of a reference monitor, offering assurance benefits.

2. The cost of maintaining multiple versions of an operating system with different access control features is reduced, making it easier to develop and maintain "trusted" variants of commodity systems, as well as to specialise OS policy for appliances, embedded devices, and smart phones.

3. Vendors of third-party systems, such as anti-virus systems, intrusion detection systems, and hardening policies, can more easily write and maintain security extensions. A well-defined interface reduces dependence on OS implementation details, and offers a "contract" for semantics such as concurrency.

4. Access control research and technology transfer is facilitated.

The MAC Framework fulfils these goals: OS vendor and third-party security extensions can be compiled into the kernel, loaded at boot time, or if permitted by the semantics of a policy, even dynamically loaded and unloaded at runtime. The framework supports a variety of access control policies, from historic information flow policies such as Biba [18] and MLS [15], rule-based labelled models such as Type Enforcement [22], to UNIX-centric hardening policies that augment, rather than supplement, existing access controls. It also provides common infrastructure needed by many policies, such as

security labelling for system subjects and objects, and policy-agnostic system calls and utilities for managing security labels. When multiple policies are loaded simultaneously, the results are deterministically composed in a useful way [1].

The argument for kernel access control extensibility is similar to the arguments made for other forms of kernel extension such as the Virtual File System (VFS) and device drivers: an initial investment to create the framework pays dividends by making code more modular, factors out common infrastructure, and facilitates customisation. For example, VFS allows the same OS APIs to access files across different file system types (such as NFS, which motivated the introduction of VFS [115]). Device driver frameworks reduce code duplication, allow new device support to be added without significant OS changes, and reduce the opportunity for errors by imposing a uniform structure. Likewise, an access control extension model enables *access control localisation*, or the adaption of a kernel's access control policy for a specific environment.

The MAC Framework's design directly addresses the concurrency vulnerabilities described in Chapter 2 by integrating access control extensions with the kernel's synchronisation model. This approach avoids multiple evaluations of arguments (syntactic or semantic), and allows policy modules atomic access to kernel structures, avoiding broad classes of time-of-check-to-time-of-use vulnerabilities endemic in system call wrappers.

The starting point for this chapter was a paper presented at the Third DARPA Information Survivability Conference and Exposition (DISCEX III) in 2003, which presented the design, implementation, and evaluation of the MAC Framework as a research technology [143]. The chapter, however, describes the framework as a production technology as shipped in late 2009, and reframes its arguments and explanation in light of a further six years of research, development, and technology transfer experience. Collaborating with companies developing products based on the MAC Framework has been a central part of my PhD research into extensible access control. Significant parts of this chapter, and all of the following chapter, build on that experience:

- Extensive real-world feedback from deployment to millions of devices has led to non-trivial refinements of approach, especially with respect to allowing policy authors control of performance versus functionality tradeoffs. DTrace probes are an example of new infrastructure added during my PhD to ease policy profiling, debugging and framework validation.

- The transition to non-experimental status, and being compiled into the default FreeBSD kernel, required adaptation to increasingly parallel CPU designs, but also overcoming concerns with binary compatibility that are key to commodity system upgrade paths, much of this considered as part of my PhD.

[1] Earlier MAC Framework designs supported configurable meta-policies for composition, but a static, predictable, and useful composition has proven adequate in practice. Neumann provides a detailed analysis of the interactions between composability and security in his CHATS final report [95].

- Aspects of the MAC Framework design have been substantially revised in order to meet new requirements brought to light by the development of new classes of policy in unanticipated environments – for example, the addition of privilege management discussed in Chapter 4, also performed as part of this PhD.

This chapter begins with a brief history of the MAC framework. It then presents the motivations and design principles for the framework, the implementation of the framework and the Biba policy module, performance evaluation of the framework, and a consideration of related research. In Chapter 4, we will explore in more detail changes in the framework's design made during its evolution from a DARPA research prototype to a widely used product, focusing on specific products and their real-world use of the framework.

3.1 History of the MAC Framework

In June, 2000, I submitted an unsolicited white paper, "Poligraph," to Dr. Douglas Maughan at DARPA, proposing a flexible access control policy framework for operating systems. The goal of the design was to revisit the relationship between the operating system kernel and its access control policies in order to facilitate research, better support development of trusted operating systems, and improve support for third-party security extensions. The opportunity to investigate the practical implementation of these ideas arose through the DARPA Composable High-Assurance Trustworthy Systems (CHATS) programme, which would fund research into open source security and security composability.

Over the next three years, I was principal investigator of the Community-Based Open Source Security (CBOSS) project at NAI Labs (later McAfee Research) that prototyped the MAC Framework on FreeBSD[2]. Table 3.1 provides a rough timeline of the evolution of the framework, first as a DARPA research project, and then as a new security technology deployed in an increasing number of open source and commercial products.

The TrustedBSD MAC Framework narrows my original Poligraph proposal by investigating a subset of the security extension problem: rather than abstracting base OS security policies (such as DAC and UNIX user isolation), the framework allows policy modules to augment or supplement the base policy. This approach leaves existing policies inlined in the kernel source, but facilitates the creation of mandatory access control policy modules, a particular interest given the limited technology transfer successes of MAC in 2000.

A central thrust of the project was the creation of reference policy modules that would validate the research approach, exercise the features of the framework, and pro-

[2]Many members of this team are thanked in the acknowledgements of this thesis: sizeable research projects such as this could not possibly be completed by any one individual!

2000	TrustedBSD Project announced with MAC design goal.
2000	"Poligraph" white paper submitted to DARPA.
2001-2004	DARPA CHATS programme; NAI Labs CBOSS project develops MAC Framework in public FreeBSD Perforce.
2002	MAC Framework merged to FreeBSD 5 development tree.
2003	Framework appears in FreeBSD 5.0 marked "experimental".
2004-2007	US Navy sponsors NAI Labs improvements to the framework, SEBSD policy, and port to Mac OS X.
2006	nCircle sponsors privilege analysis of FreeBSD kernel, framework extensions to allow privilege management.
2006	Apple ships MAC OS X Leopard desktop with MAC Framework-based sandboxing.
2007	Secure Computing Corporation contributes improvements from Sidewinder transition to FreeBSD; evaluated EAL 4+.
2007-2008	Institute of Software, Chinese Academy of Sciences static analysis studies of MAC Framework.
2008	Apple introduced MAC Framework in iPhone OS 2.0 to sandbox applications distributed via App Store.
2008	Seccuris contributes framework improvements for IPC and networking while adding Biba to monitoring service.
2009	DTrace instrumentation added to MAC Framework at Cambridge, in support of Google-sponsored TESLA project.
2009	MAC Framework upgraded to "production" feature in FreeBSD 8.0, enabled in kernel by default.
2010	Apple completes Mac OS X EAL3+ evaluation with MAC Framework enforcement; iPad ships with MAC Framework.

Table 3.1: Development and deployment of the TrustedBSD MAC Framework over ten years – from a DARPA white paper to a security technology used in routers, firewalls, desktops, servers, and even smart phones and tablet computers. My PhD research at the University of Cambridge began in 2005.

vide many open source users access to MAC policies for the first time. Initial reference modules were information flow policies grounded in trusted systems research: Bell-LaPadula multi-level security (MLS) and a fixed-label Biba policy. We also implemented Fraser's LOMAC low-watermark floating label integrity policy [45], which, while also a labeled information flow policy, requires dynamic changes to subject labels on object read, and therefore has significantly different synchronisation requirements from MLS and Biba. Previous LOMAC prototypes instrumented kernels using system call interposition, provide an opportunity for us to compare the two approaches.

As the research project proceeded, we expanded our scope to adapt SELinux's FLASK/TE [127] to FreeBSD – this was done before the inception of Linux Security Modules (LSM) [148], and established a model for how FLASK, itself an abstract security extension framework, might interact with a concrete extension system such as the MAC Framework or LSM. We also developed several UNIX-centric policies, which we felt would be of interest Internet Service Providers (ISPs), another significant FreeBSD consumer chafing under the limitations of traditional UNIX access control. UNIX-centric policies rely on existing subject and object meta-data – credentials, file ownership, and file permissions – and illustrate the flexibility of the MAC Framework in supporting differing points on the security vs. performance spectrum.

The CBOSS Project also developed general-purpose OS infrastructure components necessary to support features such as mandatory access control. These included a new Pluggable Authentication Modules (PAM) implementation, OpenPAM [125], allowing the login process to be more easily extended, and UFS2, a significant revision to the UFS file system in order to provide more reliable, semantically rich, and high-performance extended attributes to store security labels [81].

In 2004, the US Navy sponsored an adaptation of the TrustedBSD MAC Framework to Apple's relatively new Mac OS X operating system [11]. At first a research project [130], the port later matured into the security framework shipped in Mac OS X to sandbox video CODECs and other high-risk code, as well as Apple's iOS [10], to sandbox third-party applications distributed to the iPhone and iPad via Apple's AppStore.

In 2005, I began my PhD in computer security at the Computer Laboratory at the University of Cambridge, and continued my involvement in the MAC Framework project as an operating system security researcher, open source contributor, and independent contractor. This allowed me to engage in further development and technology transfer of the MAC Framework to a broad range of products, observing and participating in its adaptation to diverse environments.

Further transfer successes on FreeBSD included adoption of the MAC Framework by Juniper Networks in the JunOS SDK [70], Seccuris's instrusion monitoring products [121], nCircle's policy enforcement appliances [93], and in McAfee's high-assurance Sidewinder firewall[3] [79]. The MAC Framework design has influenced other research, including, notably, the Asbestos operating system [36], which applied the MAC Framework's notion of policy-agnostic labels to application-level policy enforcement. Detailed discussion of practical experience in deploying the MAC Framework in FreeBSD, the Mac OS X and iOS ports, and enhancement of the MAC Framework for use in nCircle's appliance, may be found in Chapter 4.

[3]Ironically, despite McAfee Research having developed the MAC Framework, the framework only entered the McAfee product line through their acquisition of Secure Computing Corporation (SCC), who adopted the framework through the FreeBSD operating system.

3.2 Past approaches

Chapter 1 tracked the history of operating system security development, from early time-sharing systems, through 1970s and 1980s microkernels, security kernels, and trusted systems, to contemporary commercial operating systems spanning servers, desktops, tablets, and phones. When work on the MAC Framework commenced in 2000, commodity operating systems eschewed microkernel design for reasons of performance, leaving them unable to adopt higher-assurance constructions such as security kernels; likewise, they were prevented by hardware from using fine-grained capabilities. However, COTS systems had consistently adopted the UNIX process model, and most enforced at least discretionary access control policies on processes. With the exception of more widespread deployment of mandatory access control, little has changed a decade later.

Operating system access control extensions generally execute in the kernel address space – this allows non-bypassable mediation and reliable security labelling of system services such as file systems and inter-process communication. Prior to extension systems such as the MAC Framework and Linux Security Modules (LSM), commodity operating security designs relied on one of three approaches to implement kernel extension: direct modification of the OS source code, system call interposition, and stackable file systems; we will briefly consider each to understand the trade-offs in selecting an extension model, as well as the types of problems that arise with these approaches.

3.2.1 Direct modification

Direct modification of existing operating system source code has been the path most often taken by vendors producing a "trusted" system. It is also one of the most effective solutions for changing kernel access control policies. This strategy either bases the work on a snapshot of an operating system release, or integrates changes into the main-line development tree. Security researchers and commercial security product developers are able to understand and modify the operating system at a fine level, as well as make changes to any part of the system that requires it. The direct modification approach has been taken by many operating systems, both OS vendors and third parties, when extending commercial operating systems (such as Trusted Solaris [129], Trusted IRIX [123], and Argus Pitbull [12]), as well as in research projects such as Badger's DTE [13].

3.2.2 System call interposition

System call interposition, discussed in detail in Chapter 2, modifies the kernel's system call table, inserting new security protections between the application and kernel service. With this approach, developers avoid changing existing source code, instead introducing new security semantics by limiting access to kernel services using wrappers.

Modules maintain authorisation structures in parallel to those maintained by the

base kernel services: prior to letting process requests reach the kernel itself, they perform their own security checks, and can limit or transform requests. On return from the system call, wrappers may also enforce post-conditions and log activities.

This strategy has been applied both with and without source code access, and can be used to introduce new security restrictions with a vendor-provided distribution. Interposition has been demonstrated for both specific policies, such as Fraser's LOMAC [45], and as a more general framework for security modification such as the Generic Software Wrapper Toolkit [46].

3.2.3 Stacked file systems

File systems store persistent data for both the operating system and applications, and as a result are common targets for security extension. Security is just one potential target of file system research requiring extensibility: reliability, namespace transformation and data transformation have all driven the development of stackable file systems. This model has been explored in detail by Heidemann and Popek in FICUS [57], and extended to a portable model spanning operating systems by Zadok and Nieh in FIST [155].

With this approach, new services are "layered" over an existing file system by wrapping operations on file system objects. In a manner similar to system call interposition, stacking permits run-time behavioural modification of a file system unanticipated by the file system author. Namespace and protection transforms can limit access to objects, or securely present objects. Data transforms can also provide cryptographic protections, presenting secure access to objects, or limiting access to compromised objects.

3.3 Limitations of past approaches

Each of these models has substantial limitations: some are inherent to extending complex systems, but others are properties of the extension mechanisms themselves. All assume either a lack of interest in the security extension from the perspective of the original OS vendor, or a commitment only to a narrow set of extensions.

3.3.1 Kernel source code access

Direct source code modification is premised on access to source code – implicit for open source systems, but historically problematic for widely-used proprietary systems. This has limited access by researchers, raised the cost of development, and required close vendor involvement in successful security extension work. Even where source access is possible for proprietary systems, intermittent dispersion of source code snapshots makes it difficult to track architectural changes, and precludes timely updates to third-party extensions. Where a vendor produces security extension products themselves, these problems may well exist within their organisation, and also discourage third parties from attempting the same.

3.3.2 Tracking vendor development

Operating systems are moving targets, even once a release has been shipped. A steady stream of major and minor updates, as well as critical security patches, risk security extensions falling out of sync with their targets; already true in 2000, this is even more the case today. This problem is especially troublesome for extension systems, such as system call interposition, that rely on a complete (and static) characterisation of the system's application binary interfaces (ABI)[4] in order to operate.

Between releases, active operating system development branches move even more quickly, with fundamental changes in internal architecture central to the creation of many new kernel services. These changes may have significant security side effects, changing assumptions about information flow, access control, or simply by changing the syntax and structure of code that has been modified or is depended on.

Other than in the area of specifically published APIs and structures (such as for file systems or device drivers), security extension authors can rely on little consistency between revisions. In addition, if a security extension relies on direct modification of operating system source code, there may be literal source code conflicts in changes to code modified by both the operating system vendor and security extension vendor, and a lag before the extended version of the OS product becomes available following an original OS vendor update.

Because of the significant change between releases, or even between service packs, any formal evaluations of the composed operating system and extension face substantial assurance challenges. Security vendors must make a complete argument for assurance not only concerning their own product, but also regarding the operating system vendor's product, requiring high levels of additional investment for only incremental operating system improvements. The burden lies entirely with the extension developer to determine that the extension will operate correctly in the new environment.

3.3.3 Concurrency and lock order in threaded kernels

As discussed in Chapter 2, wrapping techniques, such as interposition and file system stacking, introduce fundamental problems in environments supporting kernel parallelism: since the base system is unmodified, wrappers must ensure that appropriate synchronisation primitives are used to prevent time-of-check-to-time-of-use races within the kernel itself. In practice, this can require substantial duplication of work between the wrapper and the base component, as well as potential lock order violations and lock recursion (leading to deadlock).

For example, security extensions may require labels on files to provide protection

[4]Similar to an API, an ABI defines the *binary interface* between two components; typically this will involve the names and types of symbols, as well as the binary layouts of data structures, etc. API changes often imply ABI changes, and like API changes, some ABI changes are backwards compatible (i.e., adding a new interface). Less innocuous ABI changes can lead to application misbehaviour and data corruption, requiring extreme care by OS vendors as they evolve their system interfaces.

for those files. To access the labels, the system call wrapper must perform a series of namespace lookups to traverse the file hierarchy to find the target of the operation. Once the check is performed, the wrapper must release all locks on the file and namespace or risk violating the kernel's lock order when the kernel attempts to perform the lookup operation. As locks are released, the namespace and protections on objects may change, resulting in a race condition between check and use. Similar races exist for all objects supporting fine-grained locking in the kernel: locks released on target processes in signal operations will permit the label on those processes to change before the kernel performs its own lookup, locking, and protection.

Since the MAC Framework design was first sketched out in 2000, this concern has only become more pressing: dual-core servers were unusual in 2000, while today, quad-core notebook computers are commonplace. Operating system vendors have invested significant effort in the intervening years to increase OS kernel concurrency in order to exploit this new hardware, making synchronisation with an interposed security framework even more difficult.

3.3.4 Policy composition

Trusted systems are often deployed with several different kernel access control policies: UNIX discretionary access control (DAC), as well as one or more mandatory access control (MAC) policies. For example, PitBull and Trusted IRIX both ship with Biba-derived integrity policies for TCB protection, in addition to Multi-Level Security to protect the confidentiality of user data. Similarly, appliance vendors often combine vendor-specific hardening policies with base system access control features – for example, firewall and router vendors may rely on UNIX access control for base-line protection, but supplement that with product-specific security features. Several examples of policy blends are considered in Chapter 4.

Safe composition of extensions is particularly important in environments where extensions may be used to respond to new threats not anticipated at design time. For example, new security limits might be imposed to reduce exposure to a newly discovered application vulnerability. However, the problem of access control policy composition is non-trivial, and a number of problems may arise in the development and use of composed access control systems.

Literal conflicts in source code

Security extensions tend to modify the same system components in the same places, resulting in a conflict: both sets of modifications cannot be simultaneously applied. For example, both Biba and MLS instrument the same file system accesses, even though they constrain flows of information in different directions.

Functional conflicts

Security extensions can also conflict in a purely functional sense, resulting in an unusable or insecure system. Of particular concern are security extension that introduce new services or administrative interfaces, which in turn may not be properly contained by other security extensions limiting access to those interfaces. For example, naive simultaneous introduction of MLS and Biba might have each use policy-specific APIs to manage labels unprotected by the other policy. As a result, MLS would not limit the setting of Biba labels on files, and vice versa, introducing unprotected information flow channels in violation of both policies.

Fail-open composition

Security extensions may interact poorly when composed with other security extensions or applications, leading to fail-open semantics. For example, in early work to decompose system privilege in Linux, "compatibility" semantics were introduced allowing `setuid root` binaries to begin execution with less than full root privilege. In isolation, such a change seems reasonable – however, when composed with applications that assumed a `root` UID connotes full privilege, failure modes were problematic. For example, on security vulnerability resulted because `sendmail` did not check the return value of `setuid` – prior to the kernel access control model change, this was safe (albeit possibly unwise). After the change, `sendmail` did not properly detect that it had failed to drop privilege [122], allowing privilege escalation. Composition failures can have serious consequences, and are extremely difficult to reason about.

3.3.5 Financial cost of implementation

Many of technical limitations translate into increased development and maintenance cost. Of particular concern are:

- *High level of complexity* due to awkward composition of the extension and the base operating system, increasing the cost of implementation, testing, and maintenance of the extension.
- *Minimal code reuse* as different extension authors independently reproduce common infrastructure, such as file system extended attributes.
- *The moving target* nature of operating system development increases maintenance costs and can lead to significant or complete re-engineering of the same extension over successive versions.

These costs discourage security extension development by third party vendors and researchers, resulting in poor adoption of new (or even old) access control approaches beyond those present in the base operating system. In a world of constantly evolving security requirements, addressing these concerns not only reduces the costs for, but also increases the effectiveness of, security researchers and security product developers.

3.4 Designing for access control extension

The TrustedBSD MAC Framework sees its genesis in the observation that trusted operating systems efforts frequently start with existing, commercial off-the-shelf (COTS) systems. For example, Trusted Solaris and Trusted IRIX are based on the general-purpose Solaris and IRIX OS products. In addition to life-cycle assurance through improved processes and documentation, OS vendors add operational assurance features beyond the UNIX baseline. These include enhanced discretionary access control, security event auditing facilities, a decomposition of root privilege into fine-grained privileges, and mandatory access control policies such as multi-level security and the Biba integrity policy.

The MAC Framework is designed to make adding new policy models easier[5] by providing an explicit framework for access control extension – a contract between OS vendor and security extension author. As with most trusted UNIX-based designs, the MAC Framework accepts the UNIX process model, and adapts kernel structures through improvements to access control and other security functionality, rather than migrating to a security kernel. The MAC Framework design also rejects interposition, avoiding concurrency and integration problems observed with system call wrappers.

The framework is a reference monitor: explicit control and notification entry points are scattered throughout security-critical processing in the kernel, and may be instrumented by policy modules. The framework adopts a number of design philosophies from Hydra [26] including a separation of policy from mechanism and an object-centric approach in which kernel services are considered object providers, and controls are placed on method invocations. An object-centric approach facilitates implementing mandatory access control policies that are concerned with the flow of information between subjects and objects (such as MLS and Biba). The object-centric perspective also imposes an ordered view on an often disordered monolithic kernel: it is the responsibility of the kernel and MAC Framework to reconcile many of the abstraction inconsistencies arising in a highly evolutionary kernel implementation, taking care of the book-keeping in order to provide a clean model for policy authors.

The TrustedBSD MAC Framework depends on two fundamental, and essentially non-technical, premises:

1. That extending the access control model of the kernel of a commodity operating system product is a natural, desirable, and inevitable event.

2. That operating system vendors are willing to modify their products to facilitate the goal of extensibility while avoiding common extension problems – perhaps motivated by their own requirements for security extension, and perhaps to support third parties.

[5] Easier rather than easy: operating systems are complex even before you add MLS!

The research target for the MAC Framework is the OS vendor, rather than the security extension vendor – a significantly different viewpoint from that taken in most system call interposition approaches.

3.4.1 Guiding principles

The dual goals of explicit access control extensibility and engagement with commodity system vendors lead to a number of philosophical and (at times) programmatic design principles. These principles have, in turn, directed countless implementation choices in the MAC Framework that will be explored throughout the remainder of this chapter:

Do not commit to a particular access control policy. Instead, provide a framework that can support many common models. One need only inspect the broad array of continuing access control research and the variety of access control products to conclude that there is no wide consensus on a "one true policy" or even "one true policy language". The practical implication of this design principle is that policies are represented by code, and not data, in the MAC Framework – unlike systems such as type enforcement, which represents flexible policy within the confines of a policy language. With the exception of a small number of static policy characteristics, such as its disposition with respect to dynamic load and unload, policy modules implement access control decisions and state management functions entirely in C code, offering significant freedom. This allows policy models to compute results entirely dynamically, or to implement a static policy language such as found in type enforcement (and the SEBSD policy module described in Chapter 4 does precisely this).

Avoid policy-specific intrusions into kernel subsystems. Augment existing kernel services to be aware of the MAC Framework, but attempt to encapsulate policy-specific concerns entirely in policy modules. While the MAC Framework is derived from the requirements of specific policies, avoid leaking of policy-specific data representation or access control approaches outside of the policy modules. An object-centric design facilitates this, capturing many common policy needs in a clean manner.

Provide policy-agnostic infrastructure to avoid code redundancy. Many policies have common infrastructure requirements beyond the selection and provision of instrumentation points in the kernel, among which are security labelling (both ephemeral and persistent), APIs for label management, and tracing. Where possible, the MAC Framework should provide these facilities, with the dual goals of reducing work for policy authors, and also allowing policy modules to capture only access control-related logic, rather than (for example) file system storage functions for label storage. Provide policy-agnostic APIs so that applications can be security-aware without being policy-specific – for example, by providing policy-agnostic label management APIs.

Policy authors determine their own security and performance trade-offs. While the MAC Framework provides infrastructure for certain heavy-weight policy designs, such as ubiquitous labelling of network packets required by Biba and MLS, many policies may not require those features. As such, the framework should impose the costs of that

infrastructure only on policies that use expensive features, allowing policy authors to explore the design space, and allowing different sites to make different choices regarding performance, functionality, and assurance.

Support multiple simultaneous and independent policies. Most commercial trusted systems include at least two different mandatory access control policies; likewise, if base policies are represented as policy modules, third parties extending the OS will want to use the same policy interfaces to add their own policies. Where possible, provide predictable, deterministic, and ideally sensible, compositions of policies – certain policies will necessarily conflict, but the framework itself should support rather than hinder policy composition. Composition raises potentially tricky questions: for example, how should one policy control subject interactions with another policy's labels? For example: if the Biba policy marks an object as *HIGH*, then a *LOW* subject should not be able to change the MLS label on the object due as that would violate the integrity policy. The MAC Framework takes the approach that such operations can be controlled, introducing a two-phase commit process for subject and object relabel operations, which may interact with multiple policies. This is detailed in Section 3.5.6

Impose structures that facilitate assurance arguments. The MAC Framework acts as a reference monitor, a structure believed to ease assurance arguments by separating policy and mechanism, allowing those aspects of access control to be separately validated. Separation eases auditing of policy implementations, such as determining whether a policy protects all important methods on an object class, and whether the policy protects all desired classes, statically. Having a well-defined set of entry points into the framework, with well-documented semantics (such as locking) makes the framework itself easier to verify using dynamic and static analysis. Similarly, selecting a sensible and deterministic composition policy makes it easier to reason about the effects of combining policies.

Design for an increasingly concurrent operating system kernel. While specifically a security goal, the importance of multiprocessing cannot be understated, especially in light of concurrency vulnerabilities found in system call interposition. In 2000, dual-CPU systems were increasingly common; today, high-end servers and network appliances have dozens of cores, with hundred-core systems on the immediate horizon. This trend shows signs of continuing as CPU designers expose further parallelism to programmers rather than trying to mask it behind superscalar designs, leading in turn to increased exposure of concurrency to operating systems and applications. FreeBSD employs threading techniques extensively within the kernel, as well as providing threading services to user processes[6] – the MAC Framework takes advantage of tight kernel integration by aligning access control check and label maintenance with existing kernel locking, avoiding races otherwise inherent to interposition.

[6]Since FreeBSD 5.0, userspace threading has changed from an M:N model to a 1:1 model – a transition that has been largely transparent to applications utilising the pthreads API.

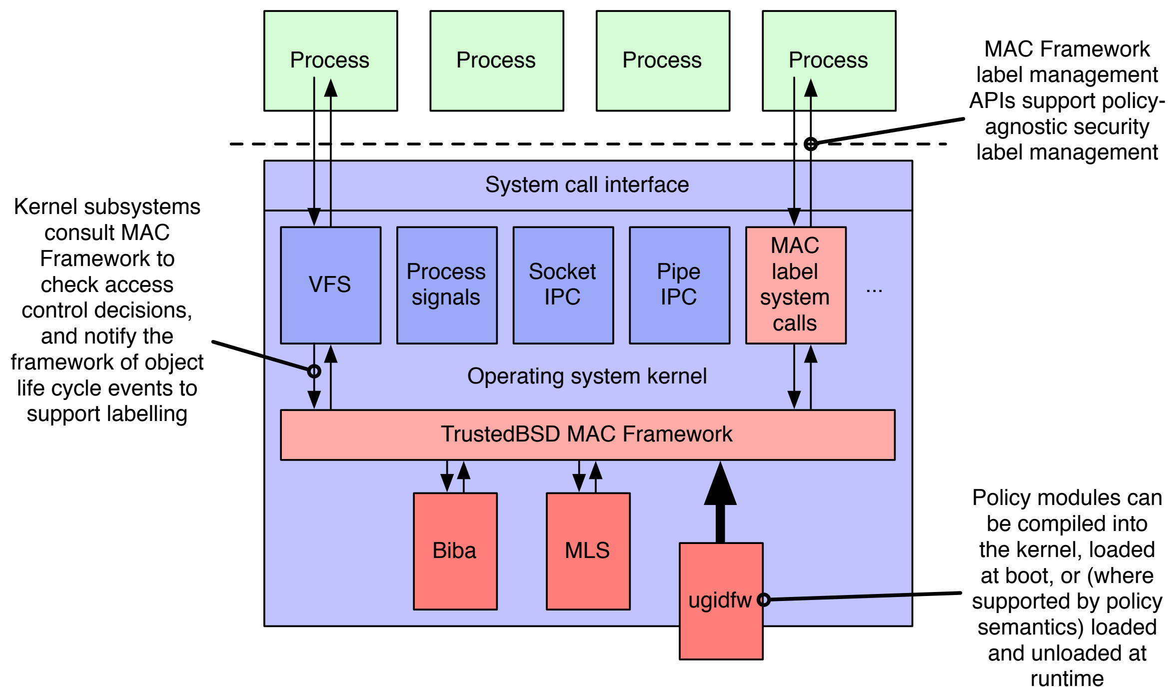

Figure 3.1: The MAC Framework defines four interfaces: the KPI from kernel services to the framework, the API from policy-agnostic user commands to the framework, the KPI between the framework and access control policy modules, and an additional debugging and tracing interface via DTrace probes not shown in this illustration.

3.5 Architecture of the MAC Framework

The MAC Framework architecture, illustrated in Figure 3.1, consists of a thin service layer linking security-aware user applications, kernel services, and access control policy modules. Policies, encapsulated in kernel modules or compiled directly into the kernel, employ the framework's infrastructure to instrument policy-relevant kernel security decisions, store and retrieve security labels on objects, handle label management requests from user applications, and dynamically compose with other loaded policies. In addition, the MAC Framework implements a set of DTrace probes, which support debugging and profiling using the D script language [24]. The framework also exposes policy-independent security-aware system calls so that monitoring and management tools can query and manipulate labels on objects.

Kernel service entry point KPI

This kernel programming interface (KPI) is called by kernel services, such as the Virtual File System (VFS) and Inter-Process Communication (IPC), to notify the MAC Framework of object events such as allocation and destruction, and to perform access control checks. Roughly 240 entry points are defined, most representing specific methods on particular object classes; generally, access control entry points take the perspective that a subject is invoking a method on an object, although this is not universally the case

– for example, IP fragment reassembly is subject-free, but security-relevant. Kernel subsystems are responsible for providing opaque storage for labels on their objects in the form of a `void *` pointer that the framework will maintain.

Policy entry point KPI

This KPI sits between the MAC Framework and registered policies. Many policy entry points correspond directly to kernel service entry points; these are supplemented by policy life cycle events, as well as a library of infrastructure functions available to policies, such as memory allocation and label storage. Policy modules need implement and invoke only those KPIs that they require: if the policy controls only process operations and not file operations, the policy will define only process-related entry points. At this layer object labels, if present, are passed directly to policy modules so that policies do not encode the layout of internal kernel data structures unless they explicitly use them.

Label management API

This application programming interface (API) allow userspace programs to query and set security labels on various objects types including files, sockets, and processes. The label management API is policy-agnostic: programs can display and manipulate labels without being aware of their specific semantics. Monitoring interfaces also allow applications to query what policies are loaded.

DTrace probes

The MAC Framework as of FreeBSD 8.0 implements a set of DTrace probes so that framework operations can be monitored using D scripts. Probes are available on the return from every MAC Framework access control entry point, and provide access to arguments and return values. These probes allow profiling and tracing of access control behaviour, which is helpful for optimising, testing, and debugging policies in both isolation and in composition.

3.5.1 Framework startup

In order to meet the non-bypassability requirements of a reference monitor, the MAC Framework must be initialised and ready to handle access control checks by the time the first user process, `init`, begins execution. Ubiquitously labelled access control policies, such as Biba and MLS, require that the framework be available significantly earlier in order to maintain security labels on all kernel objects from inception. As a result, the framework is initialised early in boot – shortly after the kernel memory allocator, console, and locking primitives become available, but before device probing and process creation have started. Initialisation occurs in several phases:

1. Framework data structures, locks, and memory allocation are initialised.
2. Policies compiled into the kernel or loaded before boot are registered.

3. The global `mac_late` flag is set, indicating that any policies loaded after this point will not be assured access to all kernel objects from inception.

4. The MAC Framework steady state is entered, and kernel boot continues.

Policies loaded after `mac_late` are not assured complete access to all events on all system objects, and are unable to rely on label memory being present for objects allocated prior to attachment. These constraints are compatible with many UNIX-centric policies, and even some labelling policies; for example, the `mac_partition` policy only needs label storage on processes created after it has been loaded. In practice, no special behaviour currently appears to be necessary at kernel shutdown. As virtualisation techniques have been applied to other sections of the kernel, the need to provide explicit shutdown for subsystem instances has become more obvious – it could be that future virtualisation work requires similar changes to the MAC Framework.

3.5.2 Policy registration

Policies must register with the MAC Framework in order to instrument access control decisions, receive object life cycle events, label object classes, and access framework services. The kernel abstraction for a "module" is distinct from the MAC Framework notion of policy: kernel modules may implement zero, one, or more independent or interdependent access control policies. This model allows the kernel, effectively a module itself, to incorporate more than one compiled-in policy at a time.

The kernel linker identifies MAC policies in the kernel and kernel modules as they are being linked using the kernel's linker set facility[7]. Each policy declares a set of properties that include whether or not the policy may be attached after boot (i.e., must it be initialised before kernel services such as process creation and the file system have started), and whether it may be unloaded. The structure of MAC Framework policies, and their registration with and use of framework infrastructure, is described in detail in Section 3.6.

When an entry point is invoked by a kernel service, the set of loaded policies is stabilised for the lifetime of that invocation; attempts to change the set of loaded policies must wait to let in-flight invocations drain before continuing. This ensures consistent implementation of access control checks, as well as preventing implementation races such as use of code in a policy after its containing module has been unloaded. In the FreeBSD 5.0 version of the MAC Framework, serialisation of entry point invocations

[7] *Linker sets* are used throughout the FreeBSD kernel to initialise and tear down module data structures and attachments to other kernel services. The facility tags static data structures during the compile-time linking process by placing them in a named ELF section; the run-time linker then iterates over sections to identify data types and function pointers requiring special handling. `SYSINIT` tags function pointers to be invoked during boot or module load, along with any ordering requirements. This approach allows identical structures to be used regardless of whether code is compiled into the kernel or loaded as a module.

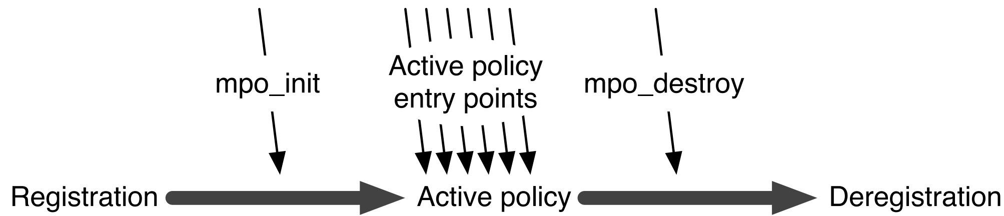

Figure 3.2: Policy life cycle: serialised initialisation and destruction entry points bracket potentially (very) concurrent entry point invocations on the active policy.

with respect to registration and deregistration was ensured by a busy count and lock, but more recent versions of the framework have relied on the read-mostly and sleepable shared-exclusive lock primitives introduced in later FreeBSD releases, which offer improved performance, and in the case of `rmlocks`, priority propagation, and starvation freedom for writers. The evolution of synchronisation in the framework, and its relationship to performance in shipping systems, is discussed in Chapter 4.

Figure 3.2 illustrates the policy life cycle: MAC policies may implement `mpo_init` and `mpo_destroy` entry points that will be invoked, respectively, during policy registration and as deregistration takes place. Both entry points are invoked while exclusive framework locks are held to ensure that all steady state entry point invocations on the policy are bracketed by the two events, allowing safe policy initialisation and cleanup.

3.5.3 Entry point design considerations

The kernel service entry point KPI is the means by which kernel subsystems, such as file systems and the network stack, engage the reference monitor in security-relevant events and decisions. Wherever possible, the MAC Framework takes the perspective that kernel subsystems implement objects whose instances may be labelled, and that policies may be adequately enforced through controls on method invocation.

This approach is often a natural fit for the kernel architecture, which often (despite a lack of formal language support for object-oriented programming) takes on an object-oriented structure. In most cases, selection of the objects to protect is a straight-forward result of analysing the APIs offered to userspace via system calls: sockets, pipes, and files, as well as the system calls that invoke methods on them seems natural. In other cases, the design choice is less clear: should all `sysctl` MBI nodes be independent objects with labels, or should they collectively be treated as a single object with read and write methods? The MAC Framework takes the latter approach on the basis that many policies require fine-grained consideration of files and directories, but not of `sysctl` nodes.

Once objects have been identified, selecting and placing entry points also requires careful design: the more granular the KPI, the more expressive policies can be – however, this is also at the cost of policy complexity. Similarly, a consistent philosophy to

placing entry point invocations is important: the fewer the invocations, the easier they are to validate – however, too few invocations leads to inadequate protection. MAC Framework entry point invocation is necessarily somewhat subjective, but generally revolves around balancing placing the checks deep enough to allow a single enforcement point for a particular level abstraction.

As an example: in early versions of the MAC Framework, access control checks for files were performed in the file systems themselves – in later versions, this was moved to the common VFS code invoking all file systems, in order to provide consistent protection. Placing VFS access control too high in the call stack for I/O system calls, however, would place them before file descriptors are differentiated into specific object types such as `vnode`s and `socket`s. File systems are necessarily involved in the storage strategy for persistent labels within the file system, but where possible rely on common infrastructure code in the MAC Framework to implement common models, such as extended attribute-based storage. Similarly, the labelling of `vnode`s rather than the on-demand provision of labels by file systems when policies make access control decisions was motivated by a desire to share abstractions across file systems, but also to provide a uniform caching model[8].

Most policy entry points are entered due to invocation of a corresponding kernel service entry point:

- Object life cycle events, such as socket creation and destruction
- Access control requests checking a subject's use of a method on an object
- General and sometimes subject-free decision requests

Entry point KPIs must be designed with great care in order to provide sufficient information so that policies can implement their semantics while also discouraging unsafe constructions that might, for example, lead to concurrency vulnerabilities. For the policy entry point KPI, it is also necessary to be sensitive to kernel binary interface (KBI) compatibility rules in the base OS, which requires that most third-party modules compiled against an earlier point release of a major release branch work on later point releases. In the MAC Framework context, that policy implies that policy modules compiled on, for example, FreeBSD 8.0 should also work on FreeBSD 8.1 to the greatest extent possible. For this reason, it is desirable to limit policy module exposure to kernel internal data structures where not specifically required for policy semantics. It is simultaneously desirable to offer the flexibility to use those internal structures where

[8]File systems operate in one of two modes in the MAC Framework: `singlelabel`, in which the underlying file system cannot support persistent storage of labels for individual files and directories, and `multilabel`, where the file system does implement this capability. In the former case, individual object labels for in-memory `vnode`s are not precluded, but the policy must tolerate loss of values when a `vnode` falls out of the cache, or an alternative method of deriving or persistently storing object labels. UFS implements `multilabel` support using extended attributes

required in order to avoid policy developers simply bypassing formal KPIs, which would be counter to the maintainability goals of the MAC Framework.

Structuring the MAC Framework to prevent bugs in policy modules, and the framework itself, is a central design concern. Where possible, the framework design employs language types to detect programmer errors; its structure also enables static analysis (such as completeness checking on controlling access to classes) through its use of symbols. Sometimes programmability and binary compatibility goals come into conflict, however. Earlier versions of the framework, prior to the advent of C99 sparse static structure initialisation, named entry points using constants registered entry point functions cast to `void *`. On face value, this approach offers stronger KBI resilience by avoiding embedding the layout of policy entry point vectors into modules – however, it also discards type information for arguments to entry points. When we experimentally switched to explicit, typed entry point functions, we discovered a number of previously unnoticed bugs in policy modules that had been incorrectly interpreting their arguments. This design choice has also been adopted by LSM, but not by Apple's kauth(9) framework, which leads to concern that policies implemented against the latter might contain similar bugs[9].

3.5.4 Kernel service entry point invocation

To understand how the MAC Framework is integrated into the kernel, and its relationship with policies, we will consider an example in the form of access control checks that occur when a file is read. An excerpt from `vn_write`, the kernel function implementing the `write` system call on files is shown in Figure 3.3. When the MAC Framework is compiled into the kernel, `vn_write` calls `mac_vnode_check_write` to authorise the request. The framework will return `0` to allow the `write` to continue, or in the event that one or more policies denies the request, a non-zero `errno` value is returned. In most cases, the MAC Framework is able to select the error number returned to userspace; this allows policies to indicate, for example, whether an error is a result of violation of a policy's rules (`EACCES`) or holds inadequate privilege (`EPERM`).

`vn_write` passes several arguments into the entry point: the credential authorising the write (`active_cred`), the credential cached in the file descriptor at the time of file open (`file_cred`), allowing MAC policies to implement UNIX-style capability rights, and the vnode that the `write` is being performed on (`vp`). The stability of arguments to entry points is ensured by the kernel synchronisation model's interaction with the calling code. Credential contents are copy-on-write, and references held by the calling thread and file descriptor prevent them from being garbage collected. The `vnode` is protected by a reference count, and `vnode` data, including the MAC label on the `vnode`,

[9] In many senses, it is weakness in the C language that forces an exchange of binary interface conservatism for type safety – programming languages implementing stronger notions of objects orientation avoid these issues by design. However, kernel programmers are, at least for the time being, stuck with C.

```
static int
vn_write(struct file *fp, struct uio *uio, struct ucred *active_cred,
    int flags, struct thread *td)
{
...
        vn_lock(vp, lock_flags | LK_RETRY);
...
#ifdef MAC
        error = mac_vnode_check_write(active_cred, fp->f_cred, vp);
        if (error == 0)
#endif
                error = VOP_WRITE(vp, uio, ioflag, fp->f_cred);
...
        VOP_UNLOCK(vp, 0);
...
        return (error);
}
```

Figure 3.3: The vnode code invokes `mac_vnode_check_write` to authorise write: two credentials, `active_cred` authorising the current operation and `file_cred` cached from file open, are passed as arguments along with the vnode `vp`.

is stabilised by the `vnode` lock; `vn_write` holds the lock over both check and use in order to ensure adequate atomicity. This construction closes several critical time-of-check-to-time-of-use races that might occur with system call interposition:

- The credential could change between check and `write`
- File labels could change between check and `write`
- The file selected by the descriptor could change between check and `write`

The arguments excluded from entry point invocation are as interesting as those included. For example, `vn_write`'s data pointer is not passed as data referenced by the pointer resides in the user address space where it cannot be accessed race-free with respect to the file write operation that will follow. These and similar design choices throughout the kernel service KPI enable access safely expressible through the kernel synchronisation model, while discouraging the expression of policies that cannot be safely represented.

3.5.5 Policy entry point invocation

The MAC Framework function `mac_vnode_check_write`, responsible for invoking policies and composing their results for `vnode` write operations, is illustrated in Figure 3.4. The function begins with a lock assertion, used during testing to ensure that the calling

```
MAC_CHECK_PROBE_DEFINE3(vnode_check_write, "struct ucred *",
    "struct ucred *", "struct vnode *");

int
mac_vnode_check_write(struct ucred *active_cred,
    struct ucred *file_cred, struct vnode *vp)
{
        int error;

        ASSERT_VOP_LOCKED(vp, "mac_vnode_check_write");
        MAC_POLICY_CHECK(vnode_check_write, active_cred, file_cred,
            vp, vp->v_label);
        MAC_CHECK_PROBE3(vnode_check_write, error, active_cred,
            file_cred, vp);
        return (error);
}
```

Figure 3.4: The MAC Framework implements `mac_vnode_check_write`: the locking protocol is validated, policies are invoked and composed using `MAC_POLICY_CHECK`, and a DTrace probe is fired on completion.

kernel service is observing the synchronisation protocol for labels and policy enforcement. Normally, assertions are compiled out of production FreeBSD kernels, but may be compiled in to provide fail-stop semantics when certain classes of synchronisation bugs are encountered.

Next, `mac_vnode_check_write` invokes `MAC_POLICY_CHECK`, which is responsible for busying the framework to stabilise the policy set, invoking each interested policy, composing the results, and invoking a DTrace probe that corresponds to the entry point. One notable difference between the kernel services KPI and the policy KPI is that the framework also passes pointers to object labels into policies, not just the objects themselves. This avoids embedding binary layout assumptions about the structure in order for a policy to access its labels. The policy framework arranges that only policies implementing an entry point are invoked – it also analyses the entry points declared by a policy in order to optimise its own behaviour, for example with respect to labelling. Discussion of the evolution of labelling optimisations and further consideration of the KBI resilience issue may be found in Chapter 4.

In addition to policy entry points mapped from MAC Framework entry points, there are several types of entry points to the policy that are not derived in the MAC Framework KPI, including policy registration and deregistration, label query and modification system calls, and the dedicated `mac_syscall`. These entry points are all invoked from within the MAC Framework, but via utility functions with similar structure and contents to `mac_vnode_check_write`.

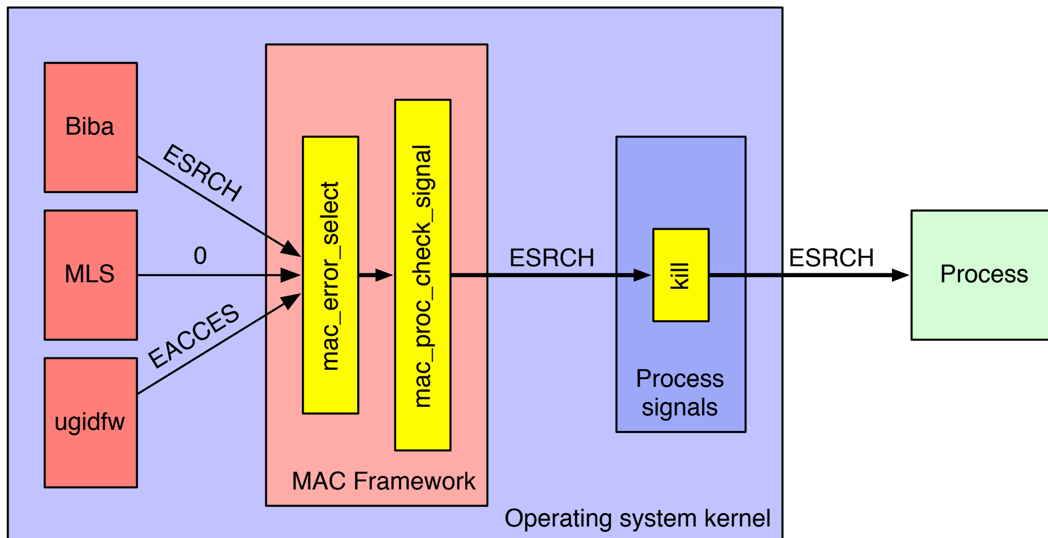

Figure 3.5: For access control checks, the results of individual policy checks are composed by `mac_error_select`, which implements a precedence operator to select which of several differing error results is the one to be returned by the system call.

3.5.6 Policy composition

Kernel entry points will invoke at least one, and possibly multiple, policy entry points. The act of policy entry point invocation is non-trivial: access to the policy list must be synchronised to prevent races with policy load and unload, the subset of policies interested in the event must have their entry point implementations called, and the results of those calls must be sensibly composed. In the original Poligraph design, this composition is controlled by a configurable meta-policy able to capture the relationship between privilege models, discretionary access control models, and mandatory access control models. In the MAC Framework design, only policies limiting the rights granted to subjects relative to the base access control policy are supported, requiring a much simpler composition in which the set of rights granted is the intersection of rights granted across all registered policies. This meta-policy is simple, deterministic, predictable by developers, and above all, useful.

Policy entry points may be broadly categorised into three types based on return type: event notifications that do not return a value, access control checks that return an `errno` value, and general decision functions that return a boolean. The composition policy requires that for an access control check to succeed, all policies expressing interest in the entry point must return success; as policies may return different error numbers in response to the same access control check, a composition function, `mac_error_select` orders and selects from among available error values. Invocation of policy entry points and composition of the results are performed using a set of composition macros[10] which

[10] A further composition macro, `MAC_POLICY_GRANT`, was added in FreeBSD 7.0 to allow policies to grant privileges; this change is detailed in Chapter 4.

combine synchronisation, selective policy invocation, and composition:

- `MAC_POLICY_PERFORM` composes policy entry points that have no return value, and is used to post events to interested policies. These events relate to policy changes, label management, policy management, or kernel object life cycle events. Since there are no return values, there is no need to provide for their composition.
- `MAC_POLICY_CHECK` composes the results of access control entry points. Unlike `MAC_POLICY_PERFORM`, it accepts an `errno` return value from each policy and composes the results using `mac_error_select`, a function that encodes an ordering of various failure classes. This model is illustrated in Figure 3.5.
- `MAC_POLICY_BOOLEAN` composes the results of decision entry points returning *true* and *false* values. It is used in scenarios where policies augment an existing kernel service decision rather than returning an access control success or failure. For example, during IP fragment reassembly, MAC policies labelling IP packets must decide if each received IP fragment matches IP fragment queues as they are iterated over. In this case, `MAC_POLICY_BOOLEAN` is used with a boolean *and*: all policies must accept a match in order for the framework to return *true*.

Some MAC Framework operations invoke more than one entry point during their operation; for example, a label set system call will need to allocate and initialise temporary label storage for the object type, copy in and internalize the userspace version of the label, perform an access control, set the label, and free the temporary storage. This sequence supports one of the more interesting aspects of policy composition: a two-phase commit on relabelling operations. This allows one policy to provide access control logic limiting the setting of labels associated with another policy on an object; for example, the Biba policy can prevent MLS labels from being set on a high-integrity object by a low-integrity subject.

3.5.7 Object labelling

Several access control policies of interest in designing the MAC Framework require additional policy-specific metadata associated with subjects (processes), and often some or all objects (files, pipes, network interfaces, ...). This metadata is referred to as a *label*, and provides subject- or object-specific information required to make access control decisions. For example, Biba labels subjects and objects with integrity levels, and MLS labels subjects with clearance information and objects with data classification levels and compartments. To this end, the MAC Framework provides a policy-agnostic label abstraction for kernel objects, system calls for querying and setting those labels (subject to control by policies), and persistent storage for labels on file system objects.

Policy modules control the contents of labels, and are responsible for imposing semantics on the label data – not just in terms of the bytes stored in labels, but also the runtime requirements for memory management, synchronisation, and persistence.

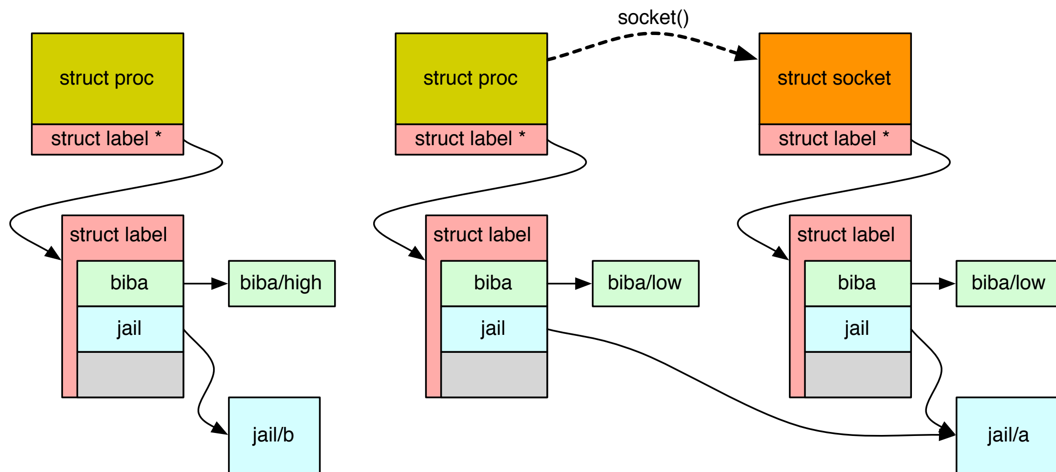

Figure 3.6: MAC labels on objects are simply managed vectors of data/pointers; policies can implement a variety of memory management models to suit their needs. In this example, the Biba policy allocates memory for each object label, whereas the Jail policy implements reference-counted, shared storage across labels.

Figure 3.6 illustrates two possible behaviours: per-object label data, and reference-counted common state across multiple objects. The framework's labelling approach is sufficiently powerful to implement these and a variety of other models that policies authors may be interested in. Providing label infrastructure addresses a number of problems, including avoiding the need for policy authors to replicate label storage facilities, and by integrating the label model with the kernel's synchronisation model in order to avoid race conditions.

Label life cycle and memory management

The MAC Framework represents label storage using `struct label`, which for policies requesting label storage, can be converted into policy-specific data. In-memory kernel data structures for labelled kernel objects, including process credentials, virtual file system nodes, and IPC objects, are extended to hold references to labels, which are managed by the MAC Framework. Table 3.2 enumerates the kernel data structures that have label storage – the top half of the table identifies objects labelled in the first release of the MAC Framework, and the bottom half are objects that became labelled in later releases. For some types, such as `struct vnode`, a label pointer is added to the data structure itself, referencing label storage allocated and managed by the framework; where kernel data structures already support a metadata scheme, such as `mbuf` tags, that facility is used to hold label data in order to avoid unnecessarily bloating the kernel data structure.

`struct label` is opaque to both kernel subsystems and MAC policies; the former invoke kernel service entry points to manage the field in the object, and the latter invoke two accessor functions, `mac_label_get` and `mac_label_set` to retrieve and set

Structure	Description
`struct bpf_d`	BPF packet sniffing device
`struct devfs_dirent`	Devfs entry
`struct ifnet`	Network interface
`struct ipq`	IP fragment queue
`struct mbuf`	In-flight packet
`struct mount`	File system mount
`struct pipe`	IPC pipe
`struct socket`	BSD IPC socket
`struct ucred`	Process credential, represents subject
`struct vnode`	VFS node: files, directories, etc
`struct inpcb`	IPv4/6 connection block
`struct ip6q`	IPv6 fragment queue
`struct ksem`	POSIX semaphore
`struct msg`	System V message
`struct msq`	System V message queue
`struct pipepair`	IPC pipe
`struct proc`	Process
`struct semid_kernel`	System V semaphore
`struct shmfd`	POSIX shared memory
`struct shmid_kernel`	System V shared memory
`struct syncache`	TCP syncache entry

Table 3.2: Kernel object types supporting MAC labels: the top half of the table lists data structures with labels added in FreeBSD 5.0; the bottom half are data structures expanded to include labels in later releases (sometimes because they reflect new features appearing in later releases).

policy-specific opaque values of type `uintptr_t`, which is sufficiently large to hold either a pointer or an integer. Internally, the MAC Framework currently implements `struct label` as an array of `uintptr_t` indexed by a per-policy *slot number* allocated on policy load if requested by the policy. However, that mechanism is left flexible due to it being entirely internal to the MAC Framework.

In FreeBSD 8.0, labels are allocated for kernel objects only when a policy specifically registers an initialisation entry point function for that object's label. This refinement was not present in earlier releases; Section 4.1.2 will discuss in greater detail the evolution of label storage. As a result of that trade-off, policies loaded after boot may find that label structures are not present for objects instantiated before the policy was loaded, and must be able to handle that case or be marked as unloadable after boot.

In the FreeBSD kernel, data structure allocation occurs in a number of forms; most frequently, the UMA slab allocator is used, which caches partially-initialised instances of objects to avoid complete reinitialisation on each reuse. In other cases, the kernel's `malloc` allocator is used, in which case full object reinitialisation occurs on each allocation. In rare cases, a subsystem manages its own memory cache in more complex ways, such as the `vnode` cache, which leaves structures fully initialised and available for continued use until the memory is reclaimed due to pressure, maximising the size of the LRU cache while still making memory logically "free". The memory model for each object is reflected in the MAC Framework and policy entry points for that object type, requiring further variation in the handling of labels across object types.

Different kernel contexts have different dispositions with respect to their ability to wait for memory to become available in the event of high memory pressure, which also affects memory allocation semantics for labels. Contexts that prevent sleeping include interrupt threads and kernel threads holding non-sleepable locks; in both of these scenarios, allowing unconditional (and hence potentially sleeping) memory allocations could lead to deadlock, and hence allocations must be allowed to fail, with knock-on effects on the complexity of calling code which must be able to handle that failure This structure is reflected in the MAC Framework: certain object initialisation calls accept an argument indicating whether sleeping is permitted; when it is forbidden, a failure to allocate a label or policy-specific label element may be returned by the policy, causing object allocation to fail. Similar constraints will exist on the allocation of the kernel object itself, and kernel subsystems will fail allocation of the containing data structure if allocation of either the `struct label` or policy-specific label storage fails.

A kernel object's label destruction is triggered when the object is destroyed by its associated kernel service. The MAC Framework is given the opportunity to release storage for the label, permitting policies to free any allocated storage or references associated with that label. Destruction may reflect the destruction of an actual object (such as a process exiting), or simply the recycling of in-memory storage for a persistent object (such as a labelled file falling out of the cache).

Kernel object creation is significantly more complex than simply allocating kernel memory: once memory is available, its fields must be initialised, including locks, it must be hooked up to namespaces, etc. Similarly, label allocation is a separate event from object creation and object association, the two mechanisms by which MAC policies may initialise their own label state given security context. *Creation* occurs when an API to create a new object is invoked: for example, a call to `open` may create a new file, `socket` a new socket, and `pipe` a new pipe. In these scenarios, the security properties of the new object (including any policy-specific MAC label data) will be initialised from sources such as the creating process's credential or the security properties of a parent object (such as a parent directory). *Association* occurs when the kernel associates an instance of a kernel data structure with an existing underlying object in persistent storage, for which the kernel data structure is simply a cache. For example, a specific file will have

a `vnode` allocated for it only after it is pulled into the in-memory working set of the file system, and may be detached from the `vnode` if it falls out of the working set and the `vnode` must be reused for another file. In that scenario, label association occurs at the point where the `vnode` is associated with the on-disk file, at which point MAC policies are given the opportunity to set up policy-specific label state, perhaps derived from the `mount` point the file is being loaded from, or from extended attributes from the on-disk file. Both the source of file label data and its interpretation are policy-specific, but the MAC Framework provides the necessary entry points to interpret and propagate label data as required. For kernel services such as the file system, creation and association operations implemented by policy modules are permitted to fail, in turn propagating failure back to the kernel service. This prevents creation of a file if, for example, storage for its security label cannot be allocated in the file system.

Label synchronisation

Where supported by the semantics of kernel locking, the MAC Framework allows policy modules to "borrow" existing kernel locks on labeled objects. This offers not only the benefit of improved performance by reducing the number of locks and locking operations, but also allows label access to be synchronised with object access, avoiding time-of-check-to-time-of-use races. Locking protocols are documented for each policy entry point, and enforced by locking assertions in debugging versions of the kernel, allowing policy developers to rely on synchronisation properties.

In some cases, however, these semantic are insufficient for policy requirements: for example, if a policy shares mutable label data between multiple objects (for example, a reference-counted sandbox descriptor), then additional synchronisation may be required to protect policy data. Similar concerns may arise where read-write locking is used on an object, and a policy needs to mutate the label (taint tracking in LOMAC, for example) while only a read lock is held by the framework; in this case, the policy must provide supplemental locking in order to ensure mutual exclusion on label data.

Another interesting case is the process credential, which itself is a reference-counted, read-only object – an important performance optimisation that reduces the memory overhead of credential data, and also allows for lock-free and thread-local use of credentials in almost all access control scenarios. When the kernel needs to modify credentials, it will perform a copy-on-write, allocating a new credential, copying old data, and modifying required fields – however, this means that much of the time, credential data is shared among not just threads, but also processes. Performing credential copy-on-write cannot be done in arbitrary contexts due to memory allocation constraints and lock order, so the LOMAC policy uses an additional process label, protected by its own locks, to tag processes for taint propagation asynchronously on the next system call return. However, in the vast majority of cases, existing object locking is sufficient to protect label data for objects. The `mac_test` module, described later in this chapter, validates that framework expectations for locking and label life cycles in entry points

are maintained. Detailed coverage of specific object behaviour can be found in the DISCEX III paper on the MAC Framework [143].

3.5.8 Application-layer approach

For the purposes of the MAC Framework, applications are divided into three categories:

Applications unaware of access control extensions require no specific adaptation. Applications may be influenced by policies as a result of new behaviour in the system (most frequently, new failures), but will not use MAC Framework system calls or APIs, such as label query interfaces. As with all security extension mechanisms, policy authors must be careful to avoid unanticipated consequences of system behaviour changes.

For example, policy authors must consider the potential impact of causing a system call to fail, where normally it would succeed unconditionally. Several security vulnerabilities have been present in shipped systems where application expectations for system behaviour were violated as the result of a "security improvements", such as the bug arising from the introduction of new privilege interfaces in Linux [122]. Prior to the addition of those interfaces, it was not possible for `getuid` to return `0` followed immediately by `setuid` failing; as a result, `sendmail` failed open, leaking root privilege.

Policy-agnostic but MAC-aware applications include traditional UNIX monitoring tools such as `ps`, `ls`, and `ifconfig`, which have been extended to display subject and object label information, but also new commands such as `setfmac` to set MAC labels on a file. The system login process has also been extended to set labels on process credentials based on user classes defined in `login.conf`. These programs all treat labels in an abstract, policy-agnostic manner. The userland framework relies on a configuration file, `/etc/mac.conf`, to determine administrator-defined defaults for labels to query and list on files, interfaces, and processes.

Policy-specific applications are aware of the specific semantics of a security policy, and if applicable, the security labels it places on objects. Depending on the nature of the the application, developers may choose to use the policy-agnostic interfaces provided by the MAC Framework, or new policy-specific interfaces exported specifically by the policy. For example, applications that are aware of the semantics of MLS labels may perform labelling operations involving only MLS label elements via policy-agnostic labelling interfaces. On the other hand, the `ugidfw` policy module exports a rule list via the kernel `sysctl` management interface.

3.5.9 Policy-agnostic label management APIs

In order to implement these functions, the kernel provides new system calls and socket options to support querying and setting labels in a policy-agnostic format, such as `mac_get_file`, `mac_get_fd`, `mac_set_file`, and `mac_set_fd`, which get and set labels on files and file descriptors. Applications handle MAC labels via the opaque `mac_t` type, which is implemented as a string buffer internally.

Labels manipulated by applications are multi-part, consisting of a series of name

and value pairs, allowing label components from different policies to be manipulated simultaneously (and with mutual atomicity) up and down the software stack. Applications can convert labels to and from an explicit text format for printing and user input; however, in general all label parsing is left up to the kernel, a design trade-off that appears acceptable, but motivated an expansion of safe string handling routines in the kernel. A typical label managed by a user application might like like `biba/low,mls/10` representing a label consisting of two elements: a low integrity biba label, and an MLS label of sensitivity "10" – applications may address all of the elements available on an object, or any subset. In earlier MAC Framework designs, we intended to allow the userspace framework for labels to be run-time extended using plugin modules, as is the case for the kernel, but this was abandoned in favour of a simpler approach.

3.6 MAC Framework policy modules

MAC Framework policies may be compiled into the kernel or encapsulated in loadable kernel modules, which themselves may be loaded during boot or at run-time. Whether a policy can be loaded or unloaded dynamically depends on its semantics, which are exposed as properties during policy registration. The MAC Framework imposes a common structure on policies: a declaration, an optional "slot" reservation for labelling if required, and a set of functions implementing declared entry points.

All but the most trivial of policies will separate entry point functions, which are aware of kernel data types and interpretations, from access control logic, which will compute decisions relative to type-independent labels or a classification of entry points by information flow directions. For example, in SEBSD, a port of FLASK and Type Enforcement to the MAC Framework, entry point functions locate the Security Identifiers (SIDs) on kernel processes and objects and pass them into FLASK's decision functions, providing a static mapping from kernel events to FLASK events, and isolating FLASK from label storage and kernel data structures

FreeBSD 5.0 shipped with several kernel modules providing a diverse set of kernel access control extensions, which have been added to both by the project and in third-party products since that release. Table 3.3 lists the sample policies included with the most recent FreeBSD 8.1 release, illustrating the diversity of supported models as well as the practicality of the approach. For the purposes of exploring the structure and implementation of policy modules, we will consider the fixed-label Biba integrity policy in further detail.

3.6.1 The Biba integrity policy

The Biba integrity policy implements a strict information flow policy limiting the impact of lower integrity subjects and objects on higher integrity subjects and objects [18]. In general, high integrity subjects are allowed to write but not read lower integrity objects, and low integrity subjects are allowed to read but not write high integrity objects.

Policy	Description
mac_biba	Hierarchal fixed-label integrity
ugidfw	"File system firewall" employing UNIX credentials and file ownership rather than labels – while the kernel module is named mac_bsdextended, we will refer to this policy by its command line tool name, ugidfw.
mac_ifoff	Interface silencing to prevent leakage in packet capture environments
mac_lomac	Hierarchal floating-label integrity
mac_mls	Multi-Level Security (BLP) with compartments
mac_none	Stub policy: all entry points are implemented as no-ops; a template for new policies
mac_partition	Inter-process visibility policy based on process partition labels assigned by the administrator
mac_portacl	TCP/IP port access control list policy, allowing the administrator to control application uses of the TCP and UDP port namespaces
mac_seeotheruids	Inter-process visibility policy based on UNIX UIDs
mac_test	MAC Framework invariant validation suite: tests locking protocols and object life cycles

Table 3.3: Table of policy modules

Biba relies on ubiquitous labelling of all system objects, and determines access control results with a dominance operator over label pairs. The policy has been frequently deployed in trusted systems to protect the Trusted Computing Base (TCB).

The TrustedBSD Biba policy module implements a hierarchal and compartmental, fixed-label mandatory integrity policy. Subject labels contain an effective label element, as well as a range of available elements; the range, set by the administrator, supports a scoped notion of privilege for users and processes by allowing selected processes to"float" up between levels or compartments for the purposes of I/O, and may also relabel objects subject to those constraints. A range with a lower value of the special constant *LOW* and a higher value of special constant *HIGH* has complete privilege within the Biba model, and is therefore also used to authorise privileged system operations that might violate the integrity properties of the process model – the role of privilege in the MAC Framework is discussed in detail in Section 4.2. Objects in the system are labeled with a single element reflecting the integrity grade and compartment of data in the object.

MAC policies are declared using the MAC_POLICY_SET macro, which accepts a set of

```
static int      biba_slot;
#define SLOT(l) ((struct mac_biba *)mac_label_get((l), biba_slot))
#define SLOT_SET(l, val) mac_label_set((l), biba_slot, (uintptr_t)(val))
...

static struct mac_policy_ops mac_biba_ops =
{
        .mpo_init = biba_init,

        .mpo_bpfdesc_check_receive = biba_bpfdesc_check_receive,
        .mpo_bpfdesc_create = biba_bpfdesc_create,
        .mpo_bpfdesc_create_mbuf = biba_bpfdesc_create_mbuf,
        .mpo_bpfdesc_destroy_label = biba_destroy_label,
        .mpo_bpfdesc_init_label = biba_init_label,

        .mpo_cred_associate_nfsd = biba_cred_associate_nfsd,
        .mpo_cred_check_relabel = biba_cred_check_relabel,
        .mpo_cred_check_visible = biba_cred_check_visible,
        .mpo_cred_copy_label = biba_copy_label,
...
};

MAC_POLICY_SET(&mac_biba_ops,   /* Policy entry point vector. */
    mac_biba,                   /* Policy short name. */
    "TrustedBSD MAC/Biba",      /* Policy long name. */
    MPC_LOADTIME_FLAG_NOTLATE,  /* Policy flags. */
    &biba_slot);                /* Label slot pointer, if required. */
```

Figure 3.7: Annotated excerpt of the Biba policy declaration.

policy properties, such as name and behavioural flags, and makes use of the FreeBSD kernel linker set facility (described in Section 3.5.2). This causes policy registration and deregistration functions to be invoked automatically whenever the containing module is loaded and unloaded, as well as to initiate allocation of labels on desired objects. Figure 3.7 illustrates an annotated version of the declaration of the FreeBSD Biba policy.

The Biba policy is a ubiquitously labeled policy that cannot be loaded after boot or unloaded, indicated by the presence of the `MPC_LOADTIME_FLAG_NOTLATE` flag and absence of the `MPC_LOADTIME_FLAG_UNLOADOK` flag. The Biba policy module generally uses entry point-specific functions for object life cycle events and access control checks, with the exception of label initialization and destruction, which are implemented using type-independent functions `biba_init_label` and `biba_destroy_label`. This is possible

```
static int
biba_vnode_check_write(struct ucred *active_cred,
    struct ucred *file_cred, struct vnode *vp, struct label *vplabel)
{
        struct mac_biba *subj, *obj;

        if (!biba_enabled || !revocation_enabled)
                return (0);
        subj = SLOT(active_cred->cr_label);
        obj = SLOT(vplabel);
        if (!biba_dominate_effective(subj, obj))
                return (EACCES);
        return (0);
}
```

Figure 3.8: Example Biba access control check for vnode write: after checking whether the policy is enabled for revocation, `mac_biba` extracts subject and object labels; if the subject label dominates the object label, information flow is permitted.

because the Biba policy imposes a uniform internal data structure for labels across all kernel data types; however, this model is not obligatory, and policies might wish, for example, to use different data structures for subjects than for objects, as done in LOMAC.

When the MAC Framework allocates label storage, the Biba policy will allocate and attach Biba-specific label storage to the framework-provided label. When objects or subjects are created, label values are inherited from the parent subject. When objects are loaded from persistent or external store, labels are assigned based on the of the storage medium, or on multilabel file systems, from system extended attributes utilised by the Biba policy. Labels on system abstractions such as special devices and network interfaces are configured using module defaults that may be tuned at boot-time or via administrative actions.

The Biba policy centralises its access control logic in functions implementing a "dominance" operator that compares two Biba elements with respect to their types, grades, and compartments to determine in what directions information may be permitted to flow between the two. Access control entry points extract subject and object labels, and then test for dominance based on categorising information flow as subject to object, object to subject, or a bidirectional flow, as illustrated in Figure 3.8. The `SLOT` macro is defined in terms of the MAC Framework's label accessor function, `mac_label_get`, which returns the value stored in the label; in the case of Biba, this is a pointer to the Biba label structure, `struct mac_biba`.

The MAC Framework successfully isolates the details of the policy implementation from the kernel services, but also isolates the details of the kernel services from the

policy implementation, reducing the risks of minor changes in one subsystem requiring gratuitous changes in the other. The MAC Framework supports the Biba policy through a generalised label management service, permitting the policy implementor to focus on the details of the policy application rather than the mechanism of instrumenting the kernel to support the policy's instrumentation requirements.

3.7 Performance evaluation

In this section, we investigate the performance of the MAC Framework and its policies. The design of the MAC Framework is intended to satisfy three closely related performance goals:

1. To minimise the performance overhead introduced by the framework when no policies are loaded.
2. To minimise the overhead of the framework itself – for example, relating to policy list synchronisation.
3. To allow policies to select their own performance trade-offs, paying only the costs of those features actually employed.

Previous sections have considered several design choices furthering these goals, from minimising MAC Framework synchronisation overhead to avoiding the costs of label allocation when unused by policies, many developed during the effort to enable the framework in the default FreeBSD kernel in 2009. These performance goals all relate to allowing administrators to select security and performance trade offs based on the needs of a site: if no policies are used, no overhead should be incurred; as policies use more complex features, such as ubiquitous labelling, greater overhead is acceptable. In many cases, the MAC Framework design successfully mitigates the costs of unused features; in other places, these costs prove more difficult to avoid. Benchmarks presented in this chapter employ two different kernel compilations:

- `GENERIC`: Default FreeBSD 8.1 kernel.
- `NOMAC`: FreeBSD 8.1 `GENERIC` kernel recompiled to disable `options MAC`[11].

Measuring the performance of the GENERIC kernel without any policies loaded allows us to investigate the overhead of the framework when inactive – a key concern in enabling `options MAC` by default in FreeBSD 8.0. However, we are also interested in the performance overheads experienced by various types of policy, conditional on their requirements and configuration:

[11] Removing `options MAC` compiles out kernel service entry point invocations and MAC Framework infrastructure from the kernel. However, even with the framework compiled out, some data structures retain label pointer fields, slightly increasing their size.

- `none`: The `mac_none` policy does not implement any entry points. This policy can be dynamically loaded and unloaded, preventing the framework from optimising out synchronisation when determining whether the policy implements an entry point or not.

- `ugidfw`: The `mac_bsdextended` policy implements a file system firewall managed by the `ugidfw` tool, enforcing an administrator-defined set of rules expressed in terms of UNIX credentials and file ownership. The policy supports constraints along the lines of, "Regardless of file permissions, *User X* is not permitted to read or write files owned by *user Y*." Like `mac_none`, it may be dynamically loaded and unloaded; unlike `mac_none`, it implements access control entry points with a real policy.

- `biba`: The `mac_biba` policy module implements Biba's fixed label integrity model [18]. The policy labels all system subjects and objects with integrity information, and controls potential paths of information flow in the system; as a result it implements most policy entry points. In this test configuration, the policy is configured for single-label file system labelling: each mounted file system has a single label, which is propagated to a file's `vnode` label when the file enters the working set.

- `biba`/ML: The `mac_biba` policy as above, but configured for multi-label operation: files have individual labels stored persistently in extended attributes, but cached with the `struct vnode` while in the working set.

We will explore performance through a set of benchmarks intended to illustrate various aspects of performance overhead imposed by the MAC Framework and its policies. As the framework is intended to control most significant operating system services, characterising the overall performance impact is difficult – instead, we consider a small number of cases intended to explore particularly interesting aspects of performance behaviour, especially as relates to design trade offs for policies.

All performance measurements have been performed on an 8-core Intel Xeon E5320 system running at 1.86GHz with 4GB of RAM, running FreeBSD 8.1's amd64 port. At the time of writing, this system represents a common multicore server configuration. In order to minimise scheduling non-determinism and other effects, the test system was kept otherwise idle, and benchmark processes were pinned to fixed CPUs. Differences were computed using Student's t-test at 95% confidence.

3.7.1 System call performance

Our first concern is the overhead imposed by the MAC Framework on system call overhead; we therefore consider a set of system call micro-benchmarks. While not illustrative of overall performance impact for complex workloads, they allow us to understand the specific costs imposed by specific MAC Framework features. All benchmarks were run by performing the target operation in a tight loop over an interval of at least 10

Test	Kernel	Time/operation	Difference	% difference
`getuid`	`NOMAC`	$0.355 \pm 0.001\mu s$	-	-
	`GENERIC`	$0.355 \pm 0.000\mu s$	$-0.000 \pm 0.000\mu s$	$-0.12\% \pm 0.10\%$
	`none`	$0.355 \pm 0.000\mu s$	$-0.000 \pm 0.000\mu s$	$-0.11\% \pm 0.10\%$
	`ugidfw`	$0.355 \pm 0.000\mu s$	$-0.000 \pm 0.000\mu s$	$-0.11\% \pm 0.10\%$
	`biba`	$0.355 \pm 0.000\mu s$	$-0.000 \pm 0.000\mu s$	$-0.11\% \pm 0.10\%$
	`biba`/ML	$0.355 \pm 0.000\mu s$	$-0.000 \pm 0.000\mu s$	$-0.11\% \pm 0.10\%$
`pread`	`NOMAC`	$1.158 \pm 0.005\mu s$	-	-
1 byte	`GENERIC`	$1.154 \pm 0.010\mu s$	†	†
	`none`	$1.240 \pm 0.001\mu s$	$0.082 \pm 0.004\mu s$	$7.08\% \pm 0.32\%$
	`ugidfw`	$1.240 \pm 0.001\mu s$	$0.083 \pm 0.004\mu s$	$7.14\% \pm 0.32\%$
	`biba`	$1.187 \pm 0.001\mu s$	$0.030 \pm 0.004\mu s$	$2.56\% \pm 0.32\%$
	`biba`/ML	$1.191 \pm 0.000\mu s$	$0.033 \pm 0.004\mu s$	$2.88\% \pm 0.31\%$
ORC	`NOMAC`	$7.207 \pm 0.013\mu s$	-	-
1 byte	`GENERIC`	$7.468 \pm 0.013\mu s$	$0.344 \pm 0.015\mu s$	$3.61\% \pm 0.17\%$
	`none`	$7.684 \pm 0.015\mu s$	$0.537 \pm 0.021\mu s$	$6.62\% \pm 0.18\%$
	`ugidfw`	$8.495 \pm 0.011\mu s$	$1.333 \pm 0.013\mu s$	$17.86\% \pm 0.16\%$
	`biba`	$7.711 \pm 0.013\mu s$	$0.508 \pm 0.014\mu s$	$6.99\% \pm 0.17\%$
	`biba`/ML	$7.658 \pm 0.013\mu s$	$0.451 \pm 0.012\mu s$	$6.25\% \pm 0.17\%$
`pread`	`NOMAC`	$207.552 \pm 0.094\mu s$	-	-
1M	`GENERIC`	$207.329 \pm 0.116\mu s$	$-0.223 \pm 0.099\mu s$	$-0.11\% \pm 0.05\%$
	`none`	$207.389 \pm 0.083\mu s$	$-0.163 \pm 0.084\mu s$	$-0.08\% \pm 0.04\%$
	`ugidfw`	$207.444 \pm 0.016\mu s$	$-0.108 \pm 0.064\mu s$	$-0.05\% \pm 0.03\%$
	`biba`	$207.461 \pm 0.025\mu s$	$-0.091 \pm 0.065\mu s$	$-0.04\% \pm 0.03\%$
	`biba`/ML	$208.121 \pm 0.076\mu s$	$0.569 \pm 0.080\mu s$	$0.27\% \pm 0.04\%$
Create +	`NOMAC`	$1030.629 \pm 1.237\mu s$	-	-
`unlink`	`GENERIC`	$1035.3566 \pm 21.061\mu s$	†	†
	`none`	$1032.984 \pm 14.660\mu s$	†	†
	`ugidfw`	$1094.781 \pm 0.746\mu s$	$64.152 \pm 0.959\mu s$	$6.22\% \pm 0.09\%$
	`biba`	$1094.864 \pm 0.217\mu s$	$64.235 \pm 0.834\mu s$	$6.23\% \pm 0.08\%$
	`biba`/ML	$1022.006 \pm 4.946\mu s$	$-8.623 \pm 3.387\mu s$	$-0.84\% \pm 0.33\%$

Table 3.4: Micro-benchmark results sorted by system call; `NOMAC` is the baseline for difference calculations. Positive differences reflect greater overhead (performance loss). † indicates no statistically significant difference.

seconds, repeating for 10 iterations. Tables 3.4 and 3.5 contain tables of system call benchmark timings.

Benchmark: getuid

This test retrieves the current process's UID using the `getuid` system call, an operation uncontrolled by the MAC Framework. The test reveals little performance difference – in fact, there is a marginal performance increase (0.12%). This change likely results from differences in global variable layout between the two different compiled versions of the kernel, and reflects the difficulty in benchmarking on contemporary hardware where cache effects may appear non-deterministic. In practice, variations of as much as 2% easily arise from these effects, leading to the reasonable conclusion that the `getuid` measurements do not reflect an actual performance change.

Benchmark: pread 1 byte

This test uses the `pread` system call to read one byte from a file cached in memory, and is intended to reveal the overhead of an access control check performed alongside a minimal kernel operation. With the MAC Framework compiled in, but no policies loaded, there is no statistically significant overhead on the `pread` benchmark.

With dynamic policies loaded, the effects of framework synchronisation are immediately visible: while neither `mac_none` nor `ugidfw` implements a `vnode` read check, iterating the policy list requires synchronisation, leading to an overhead of roughly $0.08\mu s$ (7%). In the case of VFS operations, such as the `vnode` read check, framework synchronisation consists of acquiring a sleepable `sx` lock for read, a more expensive operation than the lighter weight `rmlock` acquisition that can be used in unsleepable network paths[12]

The static `mac_biba` policy, on the other hand, does not require framework synchronisation, and experiences only a $0.03\mu s$ (2.56%) overhead; similar results apply to Biba in a multilabel configuration: $0.033\mu s$ (2.88%). While the Biba policy does implement the `vnode` read check, in the default configuration revocation of files after open is not disabled, causing it to return success immediately, meaning that this measurement largely represents the overhead of the framework invoking a statically registered policy.

Benchmark: open-read-close (ORC) 1 byte

This test expands the `pread` 1 byte test to add the `open` and `close` operations required to lookup, open, and close the file. For this test, the file is located in a local UFS file system, with a path of `/tmp/file`, requiring two lookup access control checks in addition to `open` and `read` checks:

1. `mac_check_vnode_lookup` to look up `tmp` in `/`.

[12]The `rmlock` primitives allows read acquisition touching only per-CPU variables, at the cost of making write acquisition under contention significantly more expensive – hence "read-mostly". In FreeBSD 8.1 and earlier, the `rmlock` lock type can only be used in non-sleepable scenarios – i.e., the lock cannot be held over an unbounded wait on I/O commonly found in VFS. In FreeBSD 9.0, it will also be usable in sleepable scenarios as well, making it an obvious performance optimisation for the MAC Framework going forward.

2. `mac_check_vnode_lookup` to look up `file` in `/tmp`.

3. `mac_check_vnode_open` to open `/tmp/file`.

4. `mac_check_vnode_read` to read `/tmp/file`.

This test reveals a measurable overhead of $0.344\mu s$ (3.61%) for simply compiling in the framework, likely a result of increasing cache footprint rather than direct execution overhead from the framework – a disappointingly high overhead but not one atypical of micro-benchmarks in which very small functional changes can cause cache size thresholds to be exceeded.

The dynamic `mac_none` policy imposes a 6.62% overhead due to the further costs of synchronisation without actually performing an access control check, whereas the static `mac_biba` policy imposes a 6.99% performance overhead including the cost of extracting and interpreting label data as part of an access control check.

In this test, `ugidfw`, the file system firewall, imposes the greatest overhead at 17.86%, a result of the policy's own synchronisation in checking its global (and dynamic) rule list. Currently, this synchronisation is performed using a mutex; transitioning to an `rmlock` might significantly reduce this cost. In contrast, Biba does not require synchronisation for its own data structures since labels are per-object, and it is able to borrow the kernel's existing `vnode` locking for its label data.

Benchmark: pread 1M

This test reads one megabyte from a file in cache, and is intended to explore the change in overhead as the cost of a kernel operation increases relative to the fixed cost of an access control check. As might be expected, the performance impact across all framework-enabled configurations becomes negligible in comparison to the cost of copying a megabyte of data from the file system cache to user memory – all measurements reveal an overhead less than 0.5%.

Benchmark: file create and unlink

This micro-benchmark performs three system calls: `open` with the `O_CREAT` flag to create a file and return its descriptor, `close` to release the descriptor, and `unlink` to delete the file. This sequence causes a number of framework-related operations to take place:

1. Lookup of the path `/tmp/linkfile`.

2. Authorisation, creation and labelling of the new file, including in the case of the `biba`/ML test, writing its label to disk.

3. Unlinking the file, deleting it and freeing its label.

Each benchmark loop takes around $1ms$ to complete due to the complexity of the combination. Both `ugidfw` and `mac_biba` experience a roughly $64\mu s$ overhead, or around

Test	Kernel	Time/operation	Difference	% difference
local socket	NOMAC	$2.714 \pm 0.001\mu s$	-	-
	GENERIC	$2.787 \pm 0.000\mu s$	$0.073 \pm 0.000\mu s$	$2.70\% \pm 0.02\%$
	none	$2.854 \pm 0.003\mu s$	$0.141 \pm 0.002\mu s$	$5.19\% \pm 0.07\%$
	ugidfw	$2.863 \pm 0.000\mu s$	$0.150 \pm 0.000\mu s$	$5.51\% \pm 0.02\%$
	biba	$3.348 \pm 0.001\mu s$	$0.635 \pm 0.001\mu s$	$23.39\% \pm 0.02\%$
	biba/ML	$3.344 \pm 0.000\mu s$	$0.631 \pm 0.000\mu s$	$23.24\% \pm 0.02\%$
UDP socket	NOMAC	$2.961 \pm 0.001\mu s$	-	-
	GENERIC	$2.983 \pm 0.000\mu s$	$0.022 \pm 0.001\mu s$	$0.75\% \pm 0.03\%$
	none	$3.018 \pm 0.002\mu s$	$0.057 \pm 0.001\mu s$	$1.91\% \pm 0.04\%$
	ugidfw	$3.037 \pm 0.006\mu s$	$0.076 \pm 0.004\mu s$	$2.58\% \pm 0.14\%$
	biba	$3.891 \pm 0.001\mu s$	$0.930 \pm 0.001\mu s$	$31.41\% \pm 0.03\%$
	biba/ML	$3.886 \pm 0.001\mu s$	$0.925 \pm 0.001\mu s$	$31.22\% \pm 0.03\%$

Table 3.5: Micro-benchmark results sorted by system call; NOMAC is the baseline for difference calculations. Positive differences reflect greater overhead (performance loss). † indicates no statistically significant difference.

6%. Interestingly, the biba/ML benchmark sees almost no performance change despite a presumed significant increase in file system operation complexity; this effect is likely a result of interactions with UFS soft updates, which masks the cost of creation and immediate deletion of a file by preventing either operation from going to disk – label operations are also "optimised out" in this case, avoiding any I/O overhead for what is effectively a no-op.

Benchmark: local socket

This test creates and destroys a local (UNIX) domain socket using the `socket` and `close` system calls. The MAC Framework allows policies to restrict access to create sockets – for example, to limit what protocols are available to a process, and so an access control entry point is invoked. It also allows policies to label sockets, so this sequence will involve additional memory allocation and freeing for labelled policies.

The `GENERIC` kernel experiences a modest 2.70% overhead in this test for having the MAC Framework compiled in. Both `mac_none` and `ugidfw` experience a roughly 5% overhead for synchronisation required to enter the framework once for each of the two system calls. The Biba policy, however, experiences a much higher 23.79% overhead, as the framework itself allocates and frees an additional `struct label` for the socket, and the policy also allocates and frees a `struct mac_biba`. For policies able to store their state entirely within the `intptr_t`, or able to use static data or copy-on-write data, this overhead would be significantly lower as the second allocation and free would be avoided.

Benchmark: UDP socket

This test case is nearly identical to the local socket cases, except that a UDP socket is created instead of a local domain socket. UDP sockets experience percentage-wise lower (as well as in absolute terms lower) overheads for the `GENERIC`, `mac_none`, and `ugidfw` cases due to overall greater cost to the operation, although different cache properties. However, the `mac_biba` policy occurs an even higher overhead, explainable because not only must it and the framework must allocate state at the socket layer, but also for the `inpcb` associated with each TCP/IP socket, leading to a 31% overhead. This shadowing of the `socket`-layer label offers other benefits, however: only the `inpcb` lock is required to check the label, for example, avoiding additional locking and indirections during packet receipt.

System call summary

This section has explored the performance of the MAC Framework through a system of system call micro-benchmarks. The framework design and implementation has, to varying degrees, accomplished its goals of avoiding general overhead and allowing policies to determine which feature overheads they incur. In fact, in some cases the overheads are sufficiently small that they are hard to consistently measure with the background noise present even in idle operating systems: even minor data structure layout variations change false sharing effects on cache lines.

For system calls uninstrumented by the MAC Framework, such as `getuid`, there is no measurable performance overhead. In all cases, the act of compiling the MAC Framework into the kernel leaves system call performance affected by less than 4%, and in most cases, 2% or less. Framework optimisations to avoid synchronisation overhead for "static" (i.e., loadable only at boot, and not dynamically unloadable) policies appear successful: whereas the dynamic `mac_none` policy occurs a 7.08% overhead on the `pread` 1 byte benchmark, the static `mac_biba` policy incurs only 2.56% overhead despite doing more work.

However, this analysis also shows that the synchronisation associated with dynamic policies remains a significant fraction of the cost of access control for dynamic policies – further optimisation of this case appears called for. It seems likely that optimisation schemes to avoid synchronisation overheads when no policies implement a particular entry point would be beneficial: expanding the analysis performed by the framework on loaded policies beyond when to allocated labels to include when to perform synchronisation by object class might be helpful. Such an approach might eliminate the synchronisation-related performance overhead measured on the local `socket` micro-benchmark with the `mac_none` and `ugidfw` policies have been loaded, since neither implements socket-related access control checks. Employing the `rmlock` locking primitive in additional cases may also prove beneficial, eliminating use of the more expensive `sx` lock class in the framework itself.

The MAC Framework implementation appears to have been successful in avoiding label-related memory allocation and destruction overhead for policies that do not require labels. However, the current structure for dynamic label management comes at a significant overhead, requiring at least the framework, and often also the policy, to perform additional memory allocations. Various schemes could be imagined to reduce this overhead, such as having policies expose the size of storage they require to the framework, allowing it to make a single larger allocation. The impact of the policy's own memory allocation and synchronisation schemes should not be overlooked in searching for further optimisation opportunities, however: the low cost of a Biba access control check as compared to `ugidfw` is perhaps intuitively surprising given the generally higher overhead a ubiquitously labelled and enforced policy might otherwise suggest.

System call micro-benchmarks are valuable, but as the `pread` 1 megabyte benchmark illustrates, 1-byte `read` overhead can be misleading in the bigger picture. In practice most system calls, and more generally, workloads, perform some non-trivial work, leaving object labelling and access control overhead a relatively small part of the performance story.

3.7.2 Network performance

One of the kernel subsystems most sensitive to performance, and also most likely to trigger interest in the event of performance reduction, is the network stack. In order to investigate the effect of the MAC Framework and its policies on network performance, we consider two micro-benchmarks that measure network latency and throughput over the loopback interface: `netperf`'s `UDP_RR` and `TCP_STREAM` tests [63]. Table 3.6 contains a table of benchmark measurements.

Benchmark: netperf UDP_RR

`netperf`'s `UDP_RR` test is a ping-pong test synchronously exchanging minimally-sized UDP packets between client and server processes. The test captures end-to-end latency transmitting through the socket layer, UDP, and IP, a handoff to the kernel `netisr` worker thread, followed by the input path through IP, UDP, the receive-side socket code, and delivery to userspace.

Enabling `options MAC` results in a small but measurable overhead (1.15%). Neither of the dynamic policies tested, `mac_none` and `ugidfw`, performs packet-based labelling or access control checks, but the additional framework synchronisation implied by their registration is more observable (roughly 5%). Biba experiences a 11.53% performance loss due to instrumenting several points in the transmit and receive paths:

- `mac_inpcb_create_mbuf` labels outgoing `mbuf`s based on the socket label.
- `mac_ifnet_check_transmit` checks that the `mbuf` can be transmitted over the loopback interface.

Test	Kernel	Rate	Difference	% difference
UDP_RR	NOMAC	34034 ± 344tps	-	-
	GENERIC	33643 ± 272tps	-391 ± 291tps	$-1.15\% \pm 0.86\%$
	none	32444 ± 236tps	-1590 ± 277tps	$-4.67\% \pm 0.81\%$
	ugidfw	32216 ± 173tps	-1818 ± 256tps	$-5.34\% \pm 0.75\%$
	biba	30110 ± 187tps	-3924 ± 260tps	$-11.53\% \pm 0.76\%$
	biba/ML	29945 ± 192tps	-4088 ± 238tps	$-12.01\% \pm 0.70\%$
TCP_STREAM	NOMAC	$6391 \pm 20\mu s$	-	-
	GENERIC	6359 ± 22Mb/s	-32.1 ± 20.0Mb/s	$-0.50\% \pm 0.31\%$
	none	6285 ± 19Mb/s	-106.1 ± 18.3Mb/s	$-1.66\% \pm 0.29\%$
	ugidfw	6284 ± 21Mb/s	-106.7 ± 19.4Mb/s	$-1.67\% \pm 0.30\%$
	biba	6187 ± 12Mb/s	-204.0 ± 15.6Mb/s	$-3.19\% \pm 0.24\%$
	biba/ML	6175 ± 29Mb/s	-215.6 ± 23.6Mb/s	$-3.37\% \pm 0.37\%$

Table 3.6: netperf performance benchmark results; NOMAC is the baseline for differences. Negative differences reflect reduced transactions per second (tps) or bandwidth (Mb/s) (performance loss).

- mac_ifnet_create_mbuf optionally relabels the mbuf on receipt back in the loopback interface; in the case of Biba, it will relabel the packet.

- mac_inpcb_check_deliver checks that the received mbuf can be delivered to the receiving socket.

Given the amount of work performed by the Biba policy, it is pleasantly surprising that the overhead is so low, but in part reflects the fact that other costs in loopback delivery are quite high. Over the course of loopback UDP processing, multiple threads are woken up in the delivery pipeline, multiple allocations occur, and Biba itself incurs no synchronisation overheads due to its use of object-local labelling protected by existing synchronisation.

Benchmark: netperf TCP_STREAM

netperf's TCP_STREAM test is a throughput test performed over TCP, utilising the full MTU of the interface (16K by default on the FreeBSD loopback interface). This test experiences a much lower overhead due to amortisation of access control and labelling costs over a greater amount of work performed in delivering the stream. Simply compiling in the kernel introduces an almost unmeasurable 0.50% overhead; enabling policies requiring synchronisation of entry to the framework increases this to 1.66% for mac_none and 1.67% for ugidfw; fully labelled and checked mac_biba sees roughly 3%.

Kernel	Time	Difference	% difference
`NOMAC`	143.7 ± 0.9s	-	-
`GENERIC`	143.5 ± 0.7s	†	†
`none`	143.1 ± 1.6s	†	†
`ugidfw`	144.1 ± 1.0s	†	†
`biba`	143.1 ± 1.3s	†	†
`biba` (multilabel)	155.4 ± 2.4s	11.7 ± 1.7s	$8.1\% \pm 1.2\%$

Table 3.7: Kernel compile and link performance benchmark results; `NOMAC` is the baseline for differences. Positive differences reflect greater overhead (performance loss). † indicates no statistically significant difference.

3.7.3 Kernel build performance

To supplement micro-benchmarking of system calls and slightly higher-level benchmarking of network behaviour, we consider one further benchmark: kernel build and link time. This classic benchmark incorporates a blend of compute by the compiler, file system namespace manipulation due to large numbers of files created during a compile, significant amounts of read and write I/O on the file system, but also significant OS overhead in the form of process creation and destruction, VM system activity, and inter-process communication via pipes and FIFOs. In order to mask I/O latency and better exploit the 8 cores available on the test system, the build uses up to 8 simultaneous instances of itself to parallelise compilation and linking steps. Table 3.7 contains a table of benchmark timings.

At the build macro-benchmark level, none of the kernel configurations we have considered in previous sections shows any measurable performance difference: with all the other work taking place: the cost of in-kernel labelling and enforcement is unmeasurable. However, we are able to explore one further case: extended attribute-backed label storage, which did not present a performance difference for the system call micro-benchmarks measuring `read` or `open` performance due to label caching in the `vnode` of each file. In the build benchmark, however, the additional cost of I/O for the extended attributes is noticeable, leading to an 8.14% overhead on a multilabel UFS file system.

This is at least in part due to the use of synchronous I/O to ensure that Biba security labels reach disk before the inode is hooked up to its parent directory – this prevents a race condition in which a system crash might leave a new file without adequate labelling on recovery. UFS file system employs soft updates to address the problem of synchronous I/O, reordering I/O rather than requiring software to wait synchronously before proceeding to the next operation [47]. A similar optimisation to order label I/O to disk before parent directory operations would at least mask the synchronous I/O, although not the additional I/O bandwidth required for label storage.

3.8 Related work

As described in Chapter 1, the area of operating system access control has been extensively explored by research and commercial project. Research initially focused on possible access control policies, developing models such as Bell and LaPadula's BLP/MLS confidentiality [15], Biba's integrity policy [18], Boebert's (or possibly Neumann's) Type Enforcement [22, 96], and Badger's Domain and Type Enforcement [13]. Unsatisfyingly, no single policy model has proven simple, flexible, and useful for all configurations.

This in turn has led to the popularity of more flexible models (such as TE), but also research into extensible access control models increasingly based around Anderson's reference monitor [5]. Systems such as Ott's Rule-Set Based Access Control (RSBAC) for Linux [99], based on Abrams Generalized Framework for Access Control GFAC) [2], and FLASK [78] both explore this area. Similarly, system call interposition systems have attempted to fill this gap, including Badger's Generic Software Wrappers [46].

More recently, the MAC Framework and Linux Security Modules [148] have investigated this space to great effect: by providing a reference monitor that has a close integration with kernel data structures, problems with system call interposition can be avoided while still supporting higher level abstractions such as BLP, Biba, and TE (most commonly via FLASK). Unlike LSM, the MAC Framework place a strong focus on supporting infrastructure (such as labelling semantics and policy-agnostic label system calls) and the kernel synchronisation model, offering stronger guarantees for policy authors. Apple's Kernel Authorization framework (kauth) also provides kernel extensibility with the intention of supporting anti-virus systems, and has been adopted by NetBSD [9, 35], but has proven insufficiently expressive to support mandatory protection schemes, leading Apple to also adopt the TrustedBSD MAC Framework in their Mac OS X and iOS operating systems.

3.9 Conclusion

This chapter introduced the TrustedBSD MAC Framework, an access control extension framework for the commodity FreeBSD operating system. Premised on OS vendor support for access control extensibility, the goal of the framework is to improve assurance through use of a reference monitor design, reduce the cost of access control localisation, improve OS vendor support for third-party security products such as anti-virus packages, and facilitate access control research and technology transfer. To validate the framework, we implemented a variety of access control policies, ranging from traditional MAC models such as Biba and MLS, to the research LOMAC policy, to hardening models designed around UNIX credentials and file ownership. The MAC Framework incorporates a set of guiding design principles intended to address both critical concerns with prior work and obstacles to adoption:

- Vendor lock-in to a specific access control policy is avoided while improving the

flexibility to support OS vendor security extensions.

- Policy-specific kernel changes are avoided, with the majority of MAC Framework entry points being used by more than one policy.

- Infrastructure is provided for commonly required functional dependencies, such as policy-agnostic labelling of kernel objects and label management system calls.

- Policy authors can select their own performance and functionality tradeoffs, paying only for the features they use.

- Multiple policies are composed sensibly, encouraging encapsulation of independent models in their own policy modules.

- The design facilitates assurance arguments through the structure of a reference monitor, well-defined concurrency semantics, and self-testing through locking assertions and object life cycle validation implemented in the `mac_test` module.

- Increasingly concurrent OS design is directly addressed through integration with the kernel's synchronisation model, limitations on the expression of unsafe constructs, and optimisations for multiprocessing.

- Concurrency vulnerabilities described in Chapter 2 are "designed out" by providing strong synchronisation linking access control policy authorisation, access control meta-data, and methods on protected objects.

In the next chapter, we turn to the MAC Framework's evolution from a DARPA research prototype to an access control technology integrated into a myriad of widely-deployed products. Designed for the open source FreeBSD operating system, the framework has proven surprisingly adaptable to systems as diverse as Apple's iOS operating system on the iPhone and iPad to nCircle's policy enforcement appliances. The diverse range of access control policies implemented and used with the MAC Framework validates its fundamental hypothesis: that an access control extensibility framework "designed in" to the operating system facilitates the exploration and deployment of new security technology at a time when no access control policy panacea has been found.

Chapter 4

The MAC Framework: from research to product

The previous chapter described the design and implementation of the TrustedBSD MAC Framework, a kernel access control extension framework for commodity operating systems. In this chapter, I consider the framework's transition from a DARPA research prototype to its widespread use in industry via the open-source FreeBSD operating system, and evaluate the effectiveness of the MAC Framework in validating the security extensibility hypothesis. At heart, the MAC Framework is a tool for security technology transfer, and the story of how the framework itself was transferred sheds considerable light both on the nature of and motivations for technology transfer. A number of factors have contributed to the success of the framework in open source and proprietary products:

1. *The need for new access control was pressing.* The classic UNIX model failed to meet the requirements of routers and firewalls, Internet service provider systems, and smart phones. Likewise, the threat of attack became universal due to ubiquitous networking and a proliferation of malware.

2. *Technical and structural arguments for a MAC Framework*: explicit access control extensibility was indeed the preferred way to support local security extensions, and successfully catered to a range of security and performance requirements. Access control research has continued to produce and refine policy systems, but no one policy model has proven appropriate to all environments.

3. *Hardware performance improvements increased tolerance for the overhead of new security features.* So dramatic were the hardware performance enhancements of the last decade that not only could large server systems afford access control and extensibility overhead, but so could small devices such as consumer phones, which have some of the greatest security needs.

4. *The open source technology transfer strategy proved extremely successful*, providing a community and forum for collaborative research and development, but also a

pipeline for adoption into both existing and new products relying on open source foundations.

The MAC Framework has evolved considerably since 2003. Experience has brought many refinements of approach, and in some cases, significant course adjustments. While it might be argued that the need to extend the MAC Framework to meet unanticipated needs is a failure of the approach, I argue instead that the fact that the framework was adopted, and that extensions were made within the structure of the MAC Framework, reflects its success.

Much of this evolution relates directly to adoption into products, which this chapter investigates through three case studies: the open source FreeBSD operating system for which the framework was originally developed, nCircle's FreeBSD-based IP360 security and compliance monitoring appliance, and Apple's Mac OS X operating system, used in their desktop and server products, and the iOS operating system used in their iPhone, iPod Touch, and iPad product lines. In each case, I consider the motivation for using the MAC Framework, changes made to the framework to meet the requirements of a specific product, and the types of policies deployed with the product.

4.1 FreeBSD operating system

FreeBSD is an open source operating system developed by the FreeBSD Project, an open source community, and supported by the FreeBSD Foundation, a non-profit foundation[1]. FreeBSD's origins lie in the Berkeley Software Distribution (BSD), a version of UNIX developed at the University of California at Berkeley in the 1970s and 1980s; BSD is the origin of a number of critical UNIX technologies, including the Fast File System (FFS), the Berkeley TCP/IP stack, and the Berkeley sockets API [82]. Beyond its technical success, the Berkeley open source license has proven extremely successful for technology transfer, essentially allowing unrestricted use without either license fees or a requirement to return changes to the open source project[2].

As a result, FreeBSD is used throughout industry as a platform for building online services and as a foundation for building other operating system and embedded products. Countless Internet Service Providers (ISPs) including Yahoo!, New York Internet, BitGravity, Verio-NTT, and the Internet Systems Consortium (ISC) use FreeBSD as a platform for provisioning network services across hundreds of thousands of servers. Product companies such as Juniper, Cisco, McAfee, nCircle, Sandvine, NetApp, and Apple use FreeBSD as a source of operating system technology, relying on it to varying

[1]My roles as a member of the elected FreeBSD Project core team, the project's management board, and as a member of the board of directors of the FreeBSD Foundation, have offered me considerable insight into their respective operation.

[2]The BSD license is similar to academic licenses used by a number of other major computer science research organisations, including Carnegie Mellon University and MIT.

degrees to provide core infrastructure features such as networking, file systems, security, device drivers, and POSIX APIs.

This diverse source of user requirements combined with an open development community led me to create the MAC Framework, whose basic premise is an engagement between the OS vendor and downstream consumers who require the ability to extend the security model of the operating system. In practice, the framework has seen heaviest use in appliances, embedded network devices, and consumer products; of the above list, Juniper, McAfee, and Apple rely on the MAC Framework to support their custom security models.

In this section I consider the evolution of the MAC Framework from an "experimental" feature in FreeBSD 5.0, released in 2003 before I began my PhD research, to its upgrade to a production feature in FreeBSD 8.0, released in 2009. The MAC Framework is a complex piece of software; although the framework itself is only around 8,500 lines of code, with an additional 15,000 lines of code in its reference policies, it integrates with a multi-million line kernel. This integration required modifying most major kernel subsystems, adding fields to many central kernel data structures, and making significant modifications to the system boot and login mechanisms. As the changes matured and were merged, and despite best intentions, it was inevitable that not all design tradeoffs selected during initial prototyping matched the requirements of general-purpose OS use. The transition to production status relied on several factors, including an increasing confidence in the framework's correctness, additional work to improve completeness of mediation, and responsiveness to community feedback on design and performance. The MAC Framework produced significant interest in the community, leading to a number of external contributions of code, both in terms of new features and analysis of correctness, some of which will be detailed in this section.

4.1.1 Experimental feature status

The MAC Framework was merged into the FreeBSD source tree shortly before FreeBSD 5.0, and appeared in that release as an "experimental" feature. Marking the framework as experimental had a number of implications:

- Required that the framework not be configured by default, needing an explicit administrator action to enable it.
- Required that the MAC Framework be clearly marked as experimental, notifying users of its potential incompleteness or instability.
- Accepted a reasonable likelihood of security issues in early versions of the framework. Vulnerabilities in experimental features are fixed, but do not require security advisories or binary updates implied by vulnerabilities in production features.
- Avoided programming and binary interface (API, KPI, ABI, and KBI) stability requirements: MAC Framework interfaces could be changed or removed with-

out formal deprecation processes, and with reduced sensitivity to forward and backward compatibility[3].

The goal of merging a feature before it is fully mature is to seek feedback from a ore broader community and to support early adopters who are willing to use and improve it. This applied well to the MAC Framework: as an extension framework intended to encourage experimentation with new access control policies, we argued that it could become mature only through exercise by early adopters who would establish whether it was sufficiently expressive and mature to meet their needs. Marking the framework as experimental allowed this to occur without unmet expectations of stability and completeness. We were concerned with four specific problems that we felt must be resolved before the framework could be considered production-worthy:

1. The potential performance impact was poorly understood due to the narrow set of benchmarks and analyses performed on the early prototype.

2. The binary compatibility implications for the base kernel, non-MAC kernel modules, MAC policy modules, and kernel monitoring tools were not fully understood.

3. Several less commonly used OS features (such as System V IPC) were not fully mediated, limiting the framework's effectiveness for ubiquitous information flow control policies.

4. There was no integration with the OS privilege model, preventing the MAC Framework from providing effective containment of the root user.

The previous chapter provided a brief summary of how issues of binary compatibility and feature completeness were addressed. The next section considers the evolution of performance, especially in networking. Section 4.2 discusses the MAC Framework's integration with the FreeBSD privilege model done in collaboration with nCircle.

4.1.2 Performance

FreeBSD is used in performance-sensitive environments: high-volume web servers, routers, firewalls, file servers, and more recently, low-power consumer devices where I/O and computational overhead translate into reduced battery life. As discussed in Chapter 3, this implies a number of performance goals for the framework, including minimising overhead when the framework is not in use in order to make enabling the framework in the default kernel more acceptable, minimising framework overhead when it is used, and requiring policies to only pay performance penalties for features they depend on. The previous chapter's performance analysis is based on the MAC Framework

[3]It should be observed that the FreeBSD Project's own ideas about API and binary compatibility evolved significantly over this period, as pressure mounted from users to improve application and device driver compatibility between versions.

as shipped in FreeBSD 8.1 in 2010; those results were accomplished only after several years of iterative development, profiling, and feedback from real-world use. In this section, I consider various performance optimisations introduced since 2003 in order to reach the current results.

Labelled networking

Information flow MAC policies, such as Biba and MLS, require labelling and control of all potential communication paths in the system. In most MLS extensions to UNIX systems, the practical implementation of this is labelling not only of IPC primitives directly engaged by applications (such as sockets), but also in-flight packets passing between IPC endpoints. Packet labels are typically derived from the socket or network interface where the packet originated. In FreeBSD, `struct mbuf` chains represent in-flight packets, which must therefore carry labels when policies such as Biba and MLS are used.

In the initial FreeBSD 5.0 implementation, a `struct label` was embedded in the network stacks's `struct pkthdr`, which carries meta-data for packets, such as their length, originating interface pointer, VLAN information, as well as (to a limited extent) processing state associated with adding and removing protocol headers. Semantically, this provided exactly the desired behaviour: one label for each in-flight packet; however, even when the framework was compiled out of the kernel, automatic zeroing of the 20- or 40-byte structure led to a several-percent performance overhead in per-packet processing.

In FreeBSD 5.1, the MAC Framework approach for `mbuf`s was revised to use the new `m_tag` packet meta-data facility; this allows a linked list of data structures to be hung off of in-flight packets for less commonly used features, such as IPsec and MAC. In this design, overhead for non-MAC kernels is eliminated, but a more significant overhead potentially exists for policies that employ packet labelling, as MAC labels are stored in external memory that must be allocated, freed, and indirected to for every packet. Further, if other kernel features employing `mbuf` tags are present, such as IPsec, allocation overheads are multiplied, and the number of indirections to follow the list (along with associated cache misses) is similarly increased. To mitigate this expense for MAC policies that do not employ per-packet labelling, a new policy flag was added, `MPC_LOADTIME_FLAG_LABELMBUFS`; only when at least one policy had set the flag would the MAC Framework allocate labels for `mbuf`s.

Labelling of other kernel objects

While network processing is particularly sensitive to packet labelling, the same principles also apply to other kernel object types. Somewhat lagged from the evolution of `mbuf` labelling, the MAC Framework's handling of other kernel objects has also evolved. Four considerations influenced these changes:

- reducing memory overhead when the MAC Framework is compiled out,
- changing the implementation of `struct label` without affecting the binary layout of kernel data structures,
- changing the implementation of `struct label` without affecting MAC policy modules that consume it, and
- avoiding allocating labels unused by the actual loaded set of policies, especially for object types with large standing allocations (such as `vnode`s).

To this end, in FreeBSD 5.2, all embedded instances of `struct label` were replaced with label pointers, avoiding allocation of labels for non-MAC kernels, as well as allowing the implementation of the label structure to be modified without affecting the layout of core kernel data structures. This design choice has two downsides, however: an additional indirection is added to every label access, and label structures must be independently allocated and freed, requiring additional calls to the kernel memory allocator. This trade-off is considered worthwhile, but as micro-benchmarks in Chapter 3 showed, the overhead of additional allocations is non-trivial.

An important motivation to move to an opaque and separately allocated label structure was the desire to be able to modify the size and layout of the per-policy data array in the label without changing the kernel binary interface (KBI) for loadable modules. From inception, the MAC Framework had explicitly passed label pointers as well as object pointers to MAC policies in order to avoid policies incorporating dependence on the binary layout of core kernel data structures containing labels. However, in FreeBSD 7.0, the framework KPI for policies was modified to add accessor functions, `mac_label_get` and `mac_label_set`, hiding the internal implementation of label structures from policies as well.

In FreeBSD 8.0, a general strategy of avoiding label allocation for each object type unless required by a specific policy was adopted: rather than using a policy flag for each object type, the framework determines whether labelling is required for each object type by analysing the entry points a policy implements when it registers, eliminating the need for a specific `MPC_LOADTIME_FLAG_LABELMBUFS` policy flag. If a policy registers a label initialisation function for an object type, then the MAC Framework will allocate a `struct label` for each instance of the object the kernel creates[4]. For policies compiled into the kernel or loaded at boot-time, this maintains the invariant that all instances of the object type will have label storage; for policies loaded dynamically, this guaranteed may not hold as objects may have been allocated before the policy was loaded. Rather

[4]Mac OS X further refines this approach for certain types, such as `vnode`s by allowing labels to be allocated on specific objects only when required; this model is difficult to impose on all kernel objects, however, due to policy invariants that labels always be available, as unconditional memory allocation is not possible in all kernel contexts.

than dynamically modify all outstanding object instances, which would incur significant overhead, it is instead the responsibility of the policy to handle this case.

In the case of policies used with FreeBSD, these trade-offs seem to work well: information flow policies that depend on ubiquitous labelling for correctness are already statically loaded. Dynamically loaded policies will need to be able to tolerate a lack of labels on, for example, packets instantiated before the policy loaded regardless. However, this consensus has not been accepted in all environments: McAfee's Sidewinder firewall directly embeds label information in the `struct mbuf` header in their local implementation, rather than accept additional per-packet overhead. The version of the MAC Framework in Mac OS X, forked before the introduction of automatic policy interrogation for label use, makes different trade-offs for several data types due to potential memory overhead in a largely unlabelled OS environment; this is discussed in greater detail in Section 4.3.

Where possible, the MAC Framework relies on existing object locking in the kernel, avoiding additional synchronisation to protect labels where label access aligns with object access; this reflects the practical (and intentional) reality that per-object locks are often held at the time that access control checks are performed. For most labelled policies, this works well – for example, Biba and MLS set or modify labels only on object creation or during explicit relabel operations. For LOMAC, however, writes to labels may occur when only object read locks are held, requiring additional locking.

Optimisations to FreeBSD locking over time have necessarily modified MAC Framework locking – fine-grained object locking in the kernel implies fine-grained locking in the framework, both for correctness and performance. For example, as locks in different layers of the network stack were differentiated to allow greater parallelism between processing across layers, changes were required to not just locking in the MAC Framework, but also labelling: caches of socket-layer labels were introduced in network protocol layers, such as on `struct inpcb` describing the IP-layer state for TCP and UDP connections. This avoids the need to enter the socket layer from the network protocol to check labels, as is done for other cached values in the `inpcb`.

Synchronisation overhead entering the framework

As policy modules can be loaded and unloaded dynamically, the MAC Framework provides synchronisation to ensure that modules are not unloaded while functions they contain are in execution. In FreeBSD 5.1, this synchronisation is provided by a reference count, protected by a mutex, that is incremented whenever an entry point begins executing, and decremented whenever entry points complete. Attempts to modify the list of policies, either to add or remove a policy, wait for the reference count to drain and acquire the mutex.

In FreeBSD 5.2, this approach was refined by differentiating "dynamic" policies from "static" policies: a dynamic policy can be loaded or unloaded, and hence requires synchronisation around its invocation, whereas a static policy does not. A policy is

considered static if it is loaded during boot and its policy flags indicate that it cannot be unloaded; the policy mutex is not held when iterating over statically configured policies, but is acquired to test whether dynamic policies are present, in which case the reference count is bumped (and later dropped through another acquisition of the mutex). This approach is further refined in FreeBSD 5.3 by adding a kernel option `MAC_STATIC`, which allows the set of policies to be declared fixed, preventing run-time loading and unloading, but entirely avoiding the need for synchronisation in the framework – a useful feature for routers and firewalls.

In FreeBSD 8.0, further effort was made to reduce overhead from `options MAC` to near-zero so that the framework could be included in the `GENERIC` (default) kernel. An analysis of synchronisation overhead from simply entering the framework on entry points was performed and the reference count was replaced with optimised locking primitives. To avoid deadlock, one of two locks is used depending on the type of entry point: certain entry points occur in contexts permitting unbounded sleeping (such as on slow disk I/O), but others (such as in the network stack) forbid it. The introduction of read-mostly locks in FreeBSD 8 meant that not only were atomic operations no longer required to synchronise the framework, but also also avoided writes to cache lines shared across CPUs. Combined with improvements in CPU performance and a shift to cache-centric CPU designs, the resulting overhead is minimised.

As of FreeBSD 8.0, the MAC Framework is compiled into the default kernel across all architectures; while there have been reports of overhead in certain edge cases, most measurements to date have indicated that on contemporary (cache-rich) hardware, the overhead of the framework when unused is measurably insignificant. Optimisation strategies between FreeBSD 5.0 and 8.0 changed substantially: focusing on minimising overhead for first when the framework was compiled out, and later compiled in, by default. Optimisation also changed with hardware: as the focus on instruction overhead became less important, minimising cache footprint and locking overhead became critical, reflecting a transition to multicore hardware.

4.1.3 Third-party contributions to the MAC Framework

One of the most important goals of the MAC Framework was to encourage the development of new access control policies, as well as seek contributions from the open source community and downstream consumers in improving the framework. In this section, I consider several uses of the MAC Framework in open source, academia, and industry, with a particular focus on improvements motivated by policy and correctness requirements.

SEBSD

The most significant effort at an externally maintained open source policy module is SEBSD, a port of the FLASK and Type Enforcement implementation from SELinux to FreeBSD, created at NAI Labs [78, 131]. The SEBSD project led to the introduction of

a number of new MAC Framework features, including complete mediation of previously omitted services such as System V IPC. Unfortunately, this policy module has not seen widespread use in the community, nor continuing maintenance, for two reasons:

1. Type Enforcement policies are extremely complex, and while the SELinux reference policy captured security policies for many common Linux applications also used on FreeBSD, the overhead of maintaining a port of the policy was considered too great by the community.

2. The SELinux implementation of FLASK and TE falls under the GNU Public License (GPL), an obstacle to adoption into the FreeBSD base system as a tightly integrated policy module. This concern would not prevent it being maintained by third parties and distributed as an add-on module, but that has not occurred.

The SEBSD policy module also relied on two further modifications to the MAC Framework that were not merged to FreeBSD at the time: labelling and mediation of file descriptors, and more fine-grained control of system privileges ("capabilities" in Linux terminology). The latter feature now appears in the FreeBSD kernel, described in Section 4.2, albeit expressed somewhat differently than the SELinux-based version developed in SEBSD; updating and adding file descriptor labelling and mediation would not be difficult. Concerns about license and policy authoring, then, remain the primary concern for supporting SEBSD in the future. The Kylin operating system, based on FreeBSD and discussed later in this chapter, ships with the SEBSD policy.

Digital signature verification on binaries

Christian Peron of Seccuris has developed a MAC policy module that stores and validates cryptographic checksums on application binaries and their dependencies. While `mac_chkexec` is not distributed with FreeBSD, it is available for download from the FreeBSD Perforce server.

UNIX credential control entry points

In 2005, Samy Al Bahra developed several extensions to the MAC Framework to allow policy modules to exercise greater control over the UNIX credential model. New entry points were added to control credential modifications, such as setting UIDs, GIDs, and additional groups.

Static analysis of the MAC Framework for memory safety and security

Between 2007 and 2008, while working on this PhD, I mentored and collaborated with Zhouyi Zou, a PhD student at the Institute of Software, Chinese Academy of Sciences in Beijing, as part of the Google Summer of Code programme. Zou's research explored the application of static analysis and dynamic testing to the MAC Framework to validate to memory safety and the correctness of access control hook placement [150, 151].

His study revealed a number of bugs in memory handling and incorrect enforcement, for which he submitted patches. These studies have given us significantly increased confidence in the correctness of the MAC Framework implementation, as well as its policy modules.

MAC Framework DTrace probes

As part of ongoing dynamic analysis research, to be described in a future paper, I developed DTrace extensions to the MAC Framework in 2009. Described in Chapter 3, these dynamic trace points allow debugging, profiling, and analysis of the framework and its policies using DTrace's D scripting language.

4.1.4 Additional MAC Framework consumers

In addition to nCircle's and Apple's products highlighted later in this chapter, several other FreeBSD-based products employ the MAC Framework. In most cases, the amount of public detail available is limited, but this summary is suggestive of the types of uses that the framework is seeing in industry.

McAfee Sidewinder Firewall

The high-assurance Sidewinder firewall was developed by Secure Computing Corporation (SCC) and employs the Type Enforcement policy system described in Chapter 1. Historically the firewall was based on a MAC-hardened derivative of the BSD/OS operating system. The Sidewinder product was transitioned to the FreeBSD operating system with the acquisition and subsequent cancelation of the BSD/OS product line by Wind River Systems.

In converting to FreeBSD, the MAC Framework was adopted as a means to instrument access control decisions and manage security labels, using a similar construction to that explored in SEBSD. SCC submitted one change during their FreeBSD port to add an additional access control check to authorise socket creation, but otherwise was able to use the MAC Framework unmodified for a complex and comprehensive custom security policy. In general, Sidewinder employs the MAC Framework's labelling facilities as-is, with the exception of `mbuf` labelling, where for performance reasons, direct `mbuf` header modifications have been made in order to avoid indirections. Certain availability-related, rather than access control-related, modifications continue to be maintained as local patches, rather than being merged into the MAC Framework.

Juniper Junos SDK

Juniper's Junos router operating system runs on the control planes of all Juniper routers and switches. The Junos SDK [70] allows user applications to run in the router control plane, and makes use of the MAC Framework and a custom policy module to protect the integrity of the router environment, as well as isolate router applications from one another.

Kylin operating system

Kylin is a FreeBSD-derived operating system developed at the National University of Defense Technology in China [149]. Kylin blends a new microkernel with the FreeBSD kernel, and adds a large number of functional and security enhancements, including a cryptographic file system and new access control policies. Kylin uses the MAC Framework and integrated FreeBSD access control policies, as well as SEBSD prototype type enforcement implementation and Kylin-specific policy modules.

Seccuris monitoring appliance

Seccuris provides network security monitoring as a set of appliances, analysis tools, and services [121]. The system employs the Biba integrity policy via the MAC Framework, to isolate components within monitoring appliances. Seccuris has contributed a significant number of MAC Framework improvements as a result of their experience, including bug fixes, improvements in user experience (such as the labelling of special devices), and additional controls.

4.2 nCircle IP360 monitoring appliance

nCircle Network Security, Inc., is a vendor of security and compliance monitoring tools, whose customers are as varied as Facebook, HSBC, and USAID [93]. The FreeBSD-based IP360 distributed monitoring appliance is one of nCircle's core products, and can be used to scan networks for not just versions of software with known vulnerabilities, but also vulnerable configurations, as well as noncompliance with regulations such as Sarbanes-Oxley. The IP360 uses a modified version of the FreeBSD operating system; while most security changes could be made within the framework of existing operating system access controls, customers of the IP360 requested the ability to audit all parts of the configuration and contents of the appliance directly.

While wanting to meet this customer requirement, nCircle also wanted to limit the potential for damage to the appliance's data and configuration that could be caused by compromise or misuse of the audit account. The UNIX security model, while flexible, does not easily allow the constrained delegation (and arbitrary scoping) of privilege such as the right to read all files. To address this problem, nCircle turned to a custom MAC Framework policy module, which would augment the existing UNIX policy.

The nCircle MAC policy modifies the native UNIX DAC policy to allow an *audit user* full read access to the file system and current kernel configuration, bypassing DAC protections such as file permissions. To prevent potential privilege escalation from the audit account, the MAC policy would also deny access to other important system privileges, such as the ability to write to any file, or to load a kernel module. In its original design, the MAC Framework is only able to perform a subset of the required augmentation of the base OS policy: it can prevent certain types of system access, such

as arbitrary write access, but is not able to grant new privileges. This led nCircle to seek my assistance in 2006 with enhancing the MAC Framework to add a new category of functionality: the ability to selectively grant and deny a fine-grained collection of *system privileges* using a MAC policy module.

4.2.1 What are system privileges?

Operating system privilege refers to the right to perform operations that violate of the OS security policy, including the ability to manage system settings (such as network interface addresses), the ability to bypass discretionary access control (override permissions or change file ownership), and the ability to violate integrity constraints associated with the process model and kernel protection (such as the ability to load kernel modules or enable direct I/O access from a user process). In the classic UNIX design, the kernel denies system privileges to any process not running with an effective UID of 0, the root user[5].

The UNIX userspace builds on this model, combined with setuid binaries: the `init` process starts with a UID of 0, but drops privilege by switching to other users at login. When escalation of privilege is required, setuid binaries allow a process to switch to the root user, acquiring system privilege but only via an executable that will constrain actions performed with that privilege. A system of user application privileges is built on top of the kernel model, using a combination of setuid binaries and message-passing to allow users to perform selected functions, such as changing passwords, printing to shared devices, etc.

4.2.2 System privilege extensions to the MAC Framework

For the purposes of this work, our goal was to allow a policy module to augment the kernel privilege policy in order to grant (but constrain) specific privileges for a specific user. This presented three technical concerns: how to identify and distinguish different types of privilege when exercised, how to augment granting of privilege, and how to express a policy to decide when to grant that privilege.

These problems resemble, in microcosm, the larger problem addressed by the MAC Framework: the structuring of a reference monitor for access control extensibility; it seemed a natural fit for the framework despite a departure from the design choice to limit, rather than grant, additional rights. In prior work developing the Jail security model with Kamp [65], I had explored another rethinking of root privilege, allowing root users in *jailed* processes certain violations of system access control policy, but not the ability to violate process integrity constraints. I had also previously explored implementing POSIX.1e privileges on FreeBSD, although to an unsatisfactory end: we concluded that the file-system based extended privilege model proposed in POSIX was too risky, especially in light of the Linux vulnerabilities that arose through the composition of a persisting root user and the new privilege model [122].

[5]In a few edge cases, such as process count limits, the real UID may be used instead.

In the first phase, all existing uses of the `suser` and `suser_cred` kernel functions, which check the current thread or another credential for root privilege, were analysed and replaced with new calls to `priv_check` and `priv_check_cred`, which check for specific named privileges passed via an argument rather than total privilege. Figure 4.1 illustrates a few of the roughly 200 different privileges identified in the FreeBSD kernel; this contrasts with the roughly 30 privileges used in the Linux implementation of POSIX.1e, providing a much more granular naming of privileges than are collectively captured in catch-all Linux privileges such as `CAP_SYS_ADMIN`.

Certain privileges exist in equivalence classes: for example, the right to perform raw I/O and the right to load kernel modules are effectively equivalent because either implies the other. Other privileges are effectively equivalent due to the structure of userspace above the kernel: the right to read any file connotes access to the SSH private key of the host, which might be leveraged to switch to any user – a right also granted by the privilege to change a file's UID. As the equivalencies are often unclear, and because the implications of a privilege are impacted by other aspects of the context (for example: the right to override file permissions is constrained by `chroot`, which limits the ability to name files via `open`), we distinguish privileges based on their direct effects as system methods, preferring a clear mapping of functionality into privileges, which can then be interpreted and coalesced as required for policies.

Despite sacrificing the efficient representation of privilege masks in a single 32-bit integer, this approach met the needs of the FreeBSD operating system much better: for example, exemptions to the root user model implemented in Jail could be converted from flags passed selectively to invocations of `suser_cred` to a central list of permitted privileges for jailed processes, centralising implementation of the policy.

In the second phase, the new `priv_check_cred` routine was reworked to effect an explicit composition policy for sources and limitations of privilege, with two new MAC entry points added. `mac_priv_check` takes the standard MAC entry point format, accepting a credential and a named privilege argument, and returning an error code that, if non-zero, aborts the privilege check and returns that error. `mac_priv_grant` diverges from the existing MAC Framework model in two ways. First, it allows overriding the base OS privilege policy to grant new rights, rather than restrict them, replacing the default `EPERM` that will be returned by `mac_priv_grant` when privilege is not present. Second, it uses a new policy composition macro, `MAC_GRANT` that composes errors from policies by using an alternative precedence operator that defaults to `EPERM` but allows one or more policies to replace it with a successful `0` if desired.

This approach adopts design elements from the original Poligraph proposal by allowing pluggable policy modules to supplement (or replace) the system privilege model, while still offering a sensible composition. The explicit meta-policy in `priv_check_cred` means that any MAC policy may deny privileges that would otherwise be granted by the base privilege policy or another MAC policy. Assuming that no policies deny access to the privilege (including Jail), new privileges may then be granted.

```
/*
 * The remaining privileges typically correspond to one or a small
 * number of specific privilege checks, and have (relatively) precise
 * meanings.  They are loosely sorted into a set of base system
 * privileges, such as the ability to reboot, and then loosely by
 * subsystem, indicated by a subsystem name.
 */
...
#define PRIV_ACCT        2   /* Manage process accounting. */
#define PRIV_MAXFILES    3   /* Exceed system open files limit. */
#define PRIV_MAXPROC     4   /* Exceed system processes limit. */
#define PRIV_KTRACE      5   /* Set/clear KTRFAC_ROOT on ktrace. */
#define PRIV_SETDUMPER   6   /* Configure dump device. */
#define PRIV_REBOOT      8   /* Can reboot system. */
#define PRIV_SWAPON      9   /* Can swapon(). */
#define PRIV_SWAPOFF     10  /* Can swapoff(). */
#define PRIV_MSGBUF      11  /* Can read kernel message buffer. */
#define PRIV_IO          12  /* Can perform low-level I/O. */
...
/*
 * VFS privileges.
 */
#define PRIV_VFS_READ    310 /* Override vnode DAC read perm. */
#define PRIV_VFS_WRITE   311 /* Override vnode DAC write perm. */
#define PRIV_VFS_ADMIN   312  /* Override vnode DAC admin perm. */
#define PRIV_VFS_EXEC    313 /* Override vnode DAC exec perm. */
#define PRIV_VFS_LOOKUP  314 /* Override vnode DAC lookup perm. */
...
```

Figure 4.1: Excerpt from list of system privileges supported by priv(9).

In some cases, MAC entry points already existed to perform access control checks for privileged system operations. In these cases, it was necessary to decide whether the new granular privilege model was sufficient to replace the existing kernel service entry point, or whether an entry point might still be required. The deciding factor in each case was whether additional arguments passed to the entry point to supplement the credential authorising the operation were valuable for access control purposes. This reflects a fundamental difference between the privilege model and the MAC Framework viewpoint: privileges are determined solely with respect to the subject credential, whereas most entry points are considered methods on objects, and therefore granted rights can be scoped based on the objects affected. Two example entry points illustrate this distinction:

- The entry point `mac_check_system_nfsd`, which authorised a thread to become part of the NFS server, is an example of a check entirely subsumed by the privilege system, as it accepted no additional arguments. With the introduction of `mac_check_priv`[6], this entry point has been removed.

- In contrast, the entry point `mac_check_system_swapon`, which authorises a thread to configure a particular `vnode` as a swapping target is retained, as the additional object context was key to the access control check. For example, the Biba policy enforces the requirement that a system swap device or file be labelled as `HIGH`, as a swap target may hold memory contents from processes with a variety of labels.

Finally, it was necessary to decide how new privilege functionality would be exposed: we selected to implement only privilege extensibility, and not a user model for privilege management, at that time. This means that MAC policies can adjust the privilege policy, and new policies can be written to implement new privilege services, but that system administrators are not directly presented with a role-based privilege system. These features appeared in the MAC Framework as of FreeBSD 7.0; existing MAC policies such as Biba were modified to limit access to a number of system integrity privileges when running without Biba privilege, improving resilience of the policy to a compromised root user running at lower integrity grades.

4.2.3 The nCircle MAC policy

The nCircle MAC policy builds on these features to create an audit user with the right to audit the contents of most (but not all) files on the system, the system log, and the system firewall configuration:

1. It identifies a specific UNIX user ID as the *audit user*, to which all remaining modifications of the system policy will apply.

2. The privilege-granting entry point is implemented, and grants several additional privileges to the audit user, including `PRIV_MSGBUF`, granting access to the kernel log buffer, `PRIV_VFS_READ` and `PRIV_VFS_LOOKUP`, granting the right to override permission and access control list limits on reading files and directories, and `PRIV_NETINET_IPFW_READ`, granting the right to read the active firewall configuration.

3. VFS entry points are implemented, denying write access to all objects, but also, denying read access to specific files in the file system, such as the password file. This structure, in which privileges are granted via one entry point, and then moderated via another, is another example of the continuing tension between the

[6] Due to entry point name normalization in FreeBSD 8.0, this entry point is now named `mac_priv_check`; `mac_check_system_swapon` has similarly been renamed `mac_system_check_swapon`.

concept of subject-associated privileges and the desire to constrain access scoped by object.

With privilege enhancements, the MAC Framework allows the nCircle policy module to combine controlled privilege escalation with mandatory constraints on access, meeting the needs of their product while minimising local OS modification. While nCircle's specific policy module remains closed source, they contributed my improvements to FreeBSD's privilege model and MAC Framework extensions back to the FreeBSD Project under a BSD license for FreeBSD 7.0. These enhancements of the framework allowed existing policies to be improved, as well as opening the door, through implementation of a fine-grained model for privileges, to future privilege-related security features. For example, it would now be relatively straight forward to implement an assignment of system privileges to users using role-based access control (RBAC), such as found in Solaris.

4.3 Apple's Mac OS X and iOS

In short succession, Apple released versions of its two flagship operating system products, Mac OS X Leopard for desktop and server in 2007 [11], and iPhone OS 2 for Apple's iPhone and iPod Touch in 2008 [10], incorporating the TrustedBSD MAC Framework as a reference monitor. Since then, the role of MAC Framework-based policies has grown, incorporating a variety of critical security functions. The current Mac OS X Snow Leopard release ships with four MAC policy modules:

- Sandbox provides policy-driven containment of vulnerability-prone components processing untrustworthy data. Prime examples are network server processes and video CODECs[7].

- Quarantine manages taint labels on downloaded files, tracking information such as the original web site files were downloaded from. This supports features such as user confirmation before executing a newly downloaded application.

- Parental controls prevent the execution of unapproved programs[8]; not all aspects of parental control use the framework however, as most of its constraints are application-level.

- Time Machine Safety Net protects the integrity of backup data managed by Apple's Time Machine backup system.

[7]The sandbox policy shipped in Mac OS X Leopard, iPhone OS 2.0, and iPhone OS 3.0 is referred to as Seatbelt. Sandbox is a significantly reworked version of Seatbelt, and ships with Snow Leopard and iOS 4.0.

[8]The irony of using military-inspired mandatory access control to enforce parental controls has been observed by a number of parties. Interestingly, the MAC Framework is not involved in the enforcement of Digital Rights Management (DRM) in Mac OS X.

Apple's iOS 4.1 (previously iPhoneOS) ships with two policies – Sandbox, and a MAC policy integrated with iOS's code signing facility. All of these policies across both platforms employ the MAC Framework as a reference monitor, modifying core behaviours of the operating system, and enforcing diverse policies unanticipated in our initial design of the MAC Framework.

In the following sections, I describe in greater detail the history, adaptation, and use of the MAC Framework in these Apple operating systems, as well as potential future directions.

4.3.1 SEDarwin research prototype

Apple began public beta-testing of Mac OS X in September, 2000, and the promise of a commodity desktop operating system with an open source, user-modifiable kernel was difficult to ignore. That Mac OS X's XNU kernel was based in part on FreeBSD made it particularly appealing: I already had in-progress research that might easily be made available on Mac OS X, offering a further path for technology transfer. NAI Labs, where I was working at the time, began in-house ports of my early TrustedBSD features, such as extended attributes and access control lists, to Mac OS X in order to understand the platform better [138].

XNU is a sophisticated blend of OS components from several sources: CMU's Mach 3.0 microkernel, portions of the open source FreeBSD 5.0 kernel, and numerous additional features developed at Apple. While not itself a microkernel, XNU adopts many elements from Mach, including the scheduler, IPC model, and virtual memory system. Higher-level kernel components used Mach facilities within a single kernel address space, but they are also available directly to user processes. Mach IPC is central to most Mac OS X userspace subsystems, including the window server and desktop IPC model – it is not possible to usefully explore security in Mac OS X without a careful consideration of the implications of Mach IPC.

The FreeBSD process model, IPC model, network stack, and VFS are grafted onto Mach, providing a rich POSIX programming model. XNU performance is improved by having FreeBSD kernel features in the same address space as Mach, allowing subsystems such as VFS to use pointers to refer directly to Mach VM objects. Method invocations on Mach objects can then be performed via direct function invocation, eliminating the need for message passing and context switching in countless in-kernel fast paths. These performance benefits come at the cost of lost protection between Mach and BSD, and direct exposure of BSD features to userspace rather than encapsulating them using message passing, potentially making Hydra-style interposition more difficult to implement. Apple-developed kernel components found in the first release of Mac OS X included the IOKit device driver system, Network Kernel Extension model (NKEs), and the HFS+ file system; this list that has only grown with more recent versions of the operating system.

Interesting research problems abounded: how would Mac OS X combine with secu-

rity designs requiring a whole-system view? Would we find that ideas developed by NSA and its collaborators in the DTMach [120] and DTOS [127] microkernel projects applied better or worse than the monolithic kernel design addressed by the MAC Framework? How would security features such as mandatory access control and labelling play out in such a complex runtime environment with so many effectively unmodifiable proprietary base system components and third-party applications (such as Microsoft Office)? The features that made Mac OS X so promising were also the features that made if most challenging to perform research on: a commercial, mainstream desktop vendor with in-demand applications.

With the DARPA CHATS research programme under way, and an increasingly mature MAC Framework on FreeBSD, we engaged with our research sponsors (including DARPA and the US Navy) and Apple between 2003 and 2007 to port the MAC Framework and SEBSD policy module to Mac OS X[9], a process described by Vance et al at the 2007 SELinux Symposium [130][10].

4.3.2 Adapting the MAC Framework to Mac OS X

The design of the MAC Framework reflects a detailed analysis of the FreeBSD kernel; the resulting system is tightly integrated with low-level kernel memory management and synchronisation, as well as high-level services such as the file system, IPC facilities, and network stack. A reference monitor must be aware of and able to integrate with all security-critical services in the kernel – especially, it must be able to label and control access to all system objects relevant to policies such as MLS and Biba. While our adaptation of the framework to Mac OS X was able to rely heavily on Apple's reuse of FreeBSD kernel components, the framework also required fundamental changes to reflect differences in low-level and high-level kernel features between FreeBSD and XNU. The project was performed in several stages:

1. An initial port of the framework where obvious mappings existed between the two systems, protecting BSD objects from BSD system call accesses.

2. Feature enhancements to the framework to support specific Mac OS X porting goals relating to policy expression and protection.

3. Expanded analysis to cover new services found only in Mac OS X – especially, Mach objects such as tasks and ports.

In the next few sections, I consider a few of the more interesting research problems discovered during the SEDarwin project.

[9] I was principal investigator for this project both at McAfee Research and following our acquisition by SPARTA ISSO. Although I left SPARTA in 2005 to begin my PhD, I continued my collaboration with SPARTA and Apple, engaging with Apple developers adapting the framework for use in their products.

[10] Unfortunately, only a prepublication draft of this paper is available online.

Phase 1: XNU BSD kernel components

In this phase, a number of obstacles were encountered reflecting semantic differences between the two operating systems. For example, the HFS+ file system is structured around the idea of a *disk catalogue*, in which the directory namespace is a first-class object, unlike UFS in which directories are simply a form of file supporting special operations. One implication of this difference is that Mac OS X includes several system calls that operate on the catalogue rather than on individual files, such as `getattrlist` which performs bulk retrieval of both directory listings and file attributes. HFS+'s implementation of the underlying vnode operation, `vnop_readdirattr`, did not instantiate a `vnode` for each entry in order to query these attributes – a problem for the MAC Framework that assumes file system access control is performed with respect to `vnode`s, which are labelled kernel objects. In the initial port, we disabled this optimisation, forcing a `vnode` to be allocated, a change later adopted in Mac OS X for similar reasons.

There also proved to be significant differences in the basic construction of the XNU kernel: FreeBSD relies heavily on the linker set facility described in the previous chapter. Mac OS X lacks the `SYSINIT` facility, requiring manual insertion of the MAC Framework and its policies into the boot process, as well as explicit registration of policies during module init and destroy routines.

Phase 2: MAC Framework enhancements

During development of the SEDarwin prototype, the MAC Framework was enhanced in several ways to address functional differences from FreeBSD, but also to explore feature improvements in the framework itself:

- The XNU kernel's BSD subsystem included features such as POSIX semaphores and shared memory that would arrive only later in FreeBSD. In general, these features fit naturally into the existing MAC Framework model; this code would later be backported as the FreeBSD kernel grew similar features.

- The Mac OS X port of the MAC Framework supports labelling of file descriptors as well as access control on file descriptor operations (such as `lseek`). This feature was introduced during SEBSD development, but never merged back into the base FreeBSD tree; as it was required for SEDarwin, it was subsequently included in Mac OS X.

- A further enhancement added to the Mac OS X version of the MAC Framework is explicit kernel "login contexts," an abstraction allowing labelling of an entire login session. This feature was added in order to support the notion of a more complex security context associated with the session, which would augment the security labelling of a particular process – for example, for use in a Compartmented Mode Workstation policy environment [16]. This feature is, as far as I

am aware, unexercised in both the SEDarwin TE policy and all policies shipped by Apple.

- One known problem with the design of the MAC Framework in FreeBSD is that while the policy-agnostic APIs allow userspace tools to query labels for policies without specific knowledge of their semantics, the query system calls require that the userspace tool be aware of the label element names (such as "biba"). In FreeBSD, defaults are configured using `/etc/mac.conf`; in the SEDarwin prototype, the MAC Framework was extended to allow policies to declare lists of element names that they accept. Userspace is able to query that list, in order to provide a useful default set of label elements to query.

- Mac OS X has a much more formal notion of kernel programming interface (KPI) for third-party developers; in order to conform to this design, and allow greater flexibility to change policy memory allocation models, the MAC Framework in SEDarwin provides explicit memory allocation and free interfaces that the framework translates into Mach memory allocation calls.

- As part of maturing the MAC Framework on Mac OS X, many entry point names were "normalised", or given a more consistent format. This change was later largely propagated back to FreeBSD 8.0, leading to a significant change in KPI. In general, the new names better reflect the object-method paradigm, and make it easier to predict the name of an entry point.

- The MAC Framework on XNU can control configuration of the audit subsystem, but also submit its own audit records. Control of audit was later merged to the FreeBSD version of the MAC Framework, but not the ability to submit audit records.

Phase 3: Mach tasks, ports, and services in XNU

In the third phase, coverage was expanded to include Mach objects – especially, Mach tasks and Mach ports (IPC channels). A technical issue that quickly arose in adding labelling and access control entry points for Mach objects was the order of the XNU boot. The module linker is initialised only after a number of Mach objects, including early Mach tasks and ports, have been created. As a result, policies registered "early" would still miss the creation of these early Mach objects, missing the opportunity to label them from inception. In the SEDarwin port, the creation of early Mach objects is "journalled" and then entry points for those objects are replayed once early modules have been loaded, allowing early boot objects to be labelled in order, restoring the universal labelling and mediation invariants of policies such as MLS and Biba.

As with processes in FreeBSD, Mach tasks represent the "subject" in operations on the system, invoking operations (via threads) on services. An XNU process is constructed, then, of two parts: an underlying Mach task which provides the scheduling

and virtual memory properties, and a BSD process layered above it that carries UNIX properties such as credentials, file descriptors, signal state, etc. This separation presents a fundamental philosophical problem: policies such as MLS and Biba require a unified view of interactions between subjects and objects in the system, labelling both and enforcing protections between them regardless of their "layer" in an abstraction hierarchy. Is the MAC Framework a service of the Mach layer, or the BSD layer?

Our conclusion was that, while useful in some ways, the abstraction boundary between Mach and BSD was fundamentally artificial as system subjects can directly address both layers: the MAC Framework must likewise serve both layers, and be aware of data types and semantics for both layers. In the SEDarwin port, this conflict is resolved for subjects by maintaining security labels on both the BSD process and the Mach task, but with the BSD label considered the authoritative copy; label changes are propagated to the task label when required. BSD-level operations can then be authorised against the credential label, and Mach operations are authorised against the task label, maintaining the code abstraction while satisfying the needs of policies such as Type Enforcement.

Mach ports are another case in which the microkernel origins of Mach come into conflict with the design premises of the MAC Framework. Mach ports are one-way IPC channels effectively treated as object capabilities in Mach, and unlike semantically rich IPC channels in BSD, which rely on kernel-managed namespaces such as the file system, Mach ports rely solely on userspace provision of global namespaces, typically managed by `launchd`, which provides port lookup services for applications (including the desktop IPC namespace).

Taking a leaf from the DTOS book, SEDarwin adopts an approach in which userspace applications providing services, such as the namespace, are responsible for labelling and access control, since only they have visibility into the semantics of methods invoked on ports. SEDarwin provides a *label handle* abstraction, which applications can use to query labels associated with the tasks making requests via IPC, and can also invoke the MAC Framework to authorise operations from a string-based object type and operation namespace that kernel-based policies can interpret – a significant divergence from the DTOS model, which performed policy computations in a userspace security service[11].

4.3.3 Adoption by Apple

Development of the TrustedBSD MAC Framework was motivated by a need for security policies beyond UNIX discretionary access control, in turn spawned by an explosion of new use cases for UNIX-based operating systems. Apple's Mac OS X and iOS operating systems fit both of those trends: Apple is now the single largest vendor of desktop UNIX

[11]The "security server" parlance originating in DTOS can still be found in SELinux and its SEBSD/SEDarwin derivatives even though the security server now executes as a kernel-space service rather than in a userspace task.

systems, and was among the first to deploy a UNIX operating system as the software foundation for a smart phone – both a far cry from the DEC PDP-7 for which UNIX was originally developed. Apple also experienced the same pressures felt elsewhere in industry: increasingly ubiquitous networking and a clear and present threat to even the most casual use of computers. From this perspective, creating a kernel access control extension framework for the open source OS on which Apple had built their products was quite timely.

Adoption of the MAC Framework by Apple was not assured, however. Not least, competing technologies were being simultaneously considered at Apple, motivated by similar observations about new requirements and awareness of future product directions. These included a system call interposition-based technology discussed in Chapter 2, and kauth(9) [9], a kernel authorisation framework modelled in part on ideas from the MAC Framework and targeted at anti-virus vendors. Apple found arguments about the fallibility of system call interposition convincing, and in the end adopted two different access control extension technologies: kauth for third parties such as anti-virus vendors, and the MAC Framework for Apple to use in constructing sandboxing and other stronger protections.

The potential concern regarding anti-virus vendors should not be underestimated: as explored in Chapter 3, it is easy for kernel access control extensions to incorporate strong assumptions about the implementation details of kernel services. For the FreeBSD operating system, this has proven a continuing challenge despite a liberal policy on kernel binary compatibility: compatibility is required for important kernel module types across minor, but not major, version increments. In Mac OS X, development of a well-defined Kernel Programming Interface (KPI) model, along with associated Kernel Binary Interface (KBI) model, was critical to market success: when a user upgrades their kernel and the system instantly crashes, it is the OS vendor rather than the anti-virus vendor who is blamed!

While Apple clearly does not consider the MAC Framework experimental, employing it for a variety of security-critical tasks on all of its major platforms, the framework not yet part of the public programming interface of Mac OS X. Despite this status limitation, it is entirely possible for third-party module authors to create MAC Framework policy modules on Mac OS X; for any vendor considering, for example, a mandatory access control policy such as MLS, Biba, or TE, kauth would not be an option due its far simpler approach to security extensibility.

4.3.4 The Sandbox access control policy

Apple's Mac OS X and iOS MAC Framework policy modules are not open source so we are unable to consider their implementation in detail. However, there is public documentation available for the Sandbox policy, used by native Mac OS X applications

Profile	Description
`kSBXProfileNoInternet`	TCP/IP networking is prohibited.
`kSBXProfileNoNetwork`	All sockets-based networking is prohibited.
`kSBXProfileNoWrite`	File system writes are prohibited.
`kSBXProfileNoWriteExceptTemporary`	File system writes are restricted to the temporary folder /var/tmp and the folder specified by the confstr(3) configuration variable `_CS_DARWIN_USER_TEMP_DIR`.
`kSBXProfilePureComputation`	All operating system services are prohibited.

Table 4.1: Sandbox defines a number of statically configured profiles to make sandboxing easier; applications may either uses these profiles, or define their own in a Scheme policy definition language.

and third-party applications such as Google's Chrome web browser [12].

Sandbox allows applications to voluntarily restrict their access to operating system resources such as the file system, IPC namespaces, and networking. Sandboxed processes are tagged with a sandbox profile, stored in their process credential MAC label. Profiles are expressed in a configuration language built on Scheme, and restrict access to system resources described in terms of services, methods, and in some cases, expressions matching names. Applications may select from several statically defined policies provided by Apple, listed in Table 4.1, or they may define custom policies directly in the policy language.

Public Sandbox APIs allow a process to set its label directly, or through the `sandbox-exec` helper, set policies on programs brought into execution through `execve`. Figure 4.2 shows the `common.sb` profile used by Google's Chrome web browser on Mac OS X. This policy illustrates some of the key constructs supported by Sandbox: the ability to scope operations by target object, such as limiting signal delivery to specific processes, broad controls for services such as `sysctl` and POSIX shared memory, and regular expression-based selection of files to apply specific access constraints to. In Chapter 5, I consider some of the limitations of this approach.

In addition to an enforcement mode, Sandbox supports a tracing mode that allows information to be collected on what rights an application requires; `sandboxd` is

[12]The Mac OS X Sandbox model was developed solely by Apple, and is not part of the research produced in this PhD. However, an analysis of Apple's policies and framework integration contribute significantly to both our understandings of kernel access control extensibility and the technology transfer process for the MAC Framework.

Component	Description
cvmsComAgent	part of the OpenGL JIT system
cvmsServer	part of the OpenGL JIT system
fontmover	for installing fonts
kadmind	Kerberos admin (server only)
krb5kdc	Kerberos (on all systems, client and server)
mDNSResponder	Bonjour multicast DNS service discovery
mds	Spotlight query server
mdworker	Spotlight indexer process (accepts plugins)
named	BIND DNS server (server only)
ntpd	Network Time Protocol daemon
portmap	RPC registration service
quicklook-job-creation	for quicklooks ("previews" of documents, media in finder)
quicklookd	for quicklook processes
sshd	SSH privilege separation sandbox
syslogd	System log daemon
xgridagent	xgrid (OS X server only)
xgridagent_task_nobody	xgrid: anonymous client job (server only)
xgridagent_task_sombody	xgrid: authenticated client job (server only)
xgridcontrollerd	xgrid (OS X server only)

Table 4.2: A significant number of networked Mac OS X components ship with sandboxing enabled out of the box. Some components exist on, or are only sandboxed on, Mac OS X Server.

responsible for collecting and displaying this information. This mode is not dissimilar to SELinux's permissive mode, which disables enforcements and generates policy based on the accesses performed by an application.

4.3.5 Applications constrained by Sandbox

Apple employs Sandbox to constrain a number of base OS components and bundled applications, with a particular focus on network servers; Table 4.2 lists programs that have custom profiles. In addition, Sandbox is used with statically configured profiles in several cases; one notable example is the iChat video codec, which uses a computation-only profile permitting only communication with a host process via Mach IPC.

```
;;
;; Copyright (c) 2010 The Chromium Authors. All rights reserved.
;; Use of this source code is governed by a BSD-style license that can
;; be found in the LICENSE file.
;;
; This configuration file isn't used on it's own, but instead implicity
; included at the start of all other sandbox configuration files in
; Chrome.
(version 1)
(deny default)

; Support for programmatically enabling verbose debugging.
;ENABLE_LOGGING (debug deny)

; Allow sending signals to self - http://crbug.com/20370
(allow signal (target self))

; Needed for full-page-zoomed controls - http://crbug.com/11325
(allow sysctl-read)

; Each line is marked with the System version that needs it.
; This profile is tested with the following system versions:
;     10.5.6, 10.6

; Allow following symlinks
(allow file-read-metadata)  ; 10.5.6

; Loading System Libraries.
(allow file-read-data (regex #"^/System/Library/Frameworks($|/)"))
    ; 10.5.6
(allow file-read-data (regex #"^/System/Library/PrivateFrameworks($|/)"))
    ; 10.5.6
(allow file-read-data (regex #"^/System/Library/CoreServices($|/)"))
    ; 10.5.6

; Needed for IPC on 10.6
;10.6_ONLY (allow ipc-posix-shm)
```

Figure 4.2: Sample sandbox policy file included with Google's Chrome web browser.

4.3.6 Enforcement in Mach and BSD

As explored earlier in our consideration of SEDarwin, comprehensive access control policies on Mac OS X must take into account the operating system's split personality: a set of semantically strong services with well-defined, kernel-controlled namespaces in BSD, and a message-passing application model built on Mach IPC. Sandbox likewise provides two sorts of protection: access control for kernel-implemented objects such as the file system and network stack, and access control on critical userspace services reached via Mach IPC. As such, Sandbox relies primarily on kernel enforcement, but also some in the trusted `launchd` process, which implements the Mach namespace used by applications to look up Mach IPC-based services in Mac OS X.

In Seatbelt, programs setting up sandboxes would request that `launchd` create a new name service port with policy bound to it, which would be passed to the sandboxed application. In Sandbox, the kernel policy provides a policy lookup service to `launchd`, which queries policy via the `mac_syscall` policy system call, passing the PID of the requesting process (extracted from the Mach message trailer of the request). This capability-oriented approach does not require kernel enforcement in Mach IPC, a fact relied on for performance optimisations described later in this section; however, it does resemble the approach taken in DTMach and DTOS, in which policy queries could be sent to the security server based on Mach trailer tagging of remote task domain, for enforcement by arbitrary server processes.

Kernel-enforced policies are set on processes using MAC Framework process labels, which may be set explicitly on the current process, or by passing the label to `mac_execve`, a version of the `execve` call that permits the caller to request a policy-specific label transition. In Seatbelt, profile descriptions written in Scheme are submitted to the kernel policy, which will then upcall to `sandboxd` to convert the policy to an executable byte code. With Sandbox, policy is compiled to byte code by a library executing within the program creating a sandbox, eliminating the need for an up-call. Both models avoids the complexity of compiling regular expressions in the kernel, as well as avoiding the bytecode becoming part of the fixed binary interface for applications.

4.3.7 Paths in policy expression

Path-based controls are one of the highlights of the Sandbox policy model; as with DTE [13], the benefits of a path-centric approach to manually administered file labels are clear. Programmers write applications using paths to access objects, and writing policy using paths is a more accessible approach than managing individual labels on countless files, long a criticism of labelled policies such as Biba, MLS, and TE.

In FreeBSD, such a scheme would be difficult to implement correctly: the UNIX VFS model treats pathnames as second class objects – ephemeral instructions provided by the user process for traversing the actual objects: directories and files. The UNIX file system makes path-based reasoning extremely difficult: files can have zero names

(unlinked files that are still open), one name (typical files), or many names (hard linked files, and in Mac OS X, even hard linked directories). Further, one name might refer to many potential files due to `chroot` and covering effects from file system mountpoints. Names on objects can be changed without any direct access to the object itself, and the UNIX VFS model makes this occurrence difficult or impossible to efficiently track; for example, while the path used to access a hard linked file can be cached, if that name is unlinked, other names may still be valid for the file but would require a linear search of all directories in the file system to identify.

In Mac OS X, the file system namespace is semantically much stronger. The HFS+ file system implements a parent pointer from files back to their containing directories, with additional book-keeping to handle hard links. The name cache ensures that all names leading to an object are available at all times, and Mac OS X avoids complex namespace manipulations that can lead to pathname confusion, such as layered file system mounts. As a result, conversion from an object reference to a path is considered a reliable operation. The Sandbox policy also attempts to enforce operations only at initial reference, such as during file open, rather than potentially later on operations such as `read` and `write`, avoiding the need for permission evaluation both in the fast path and when paths may be less readily available.

4.3.8 Considerations for iOS

Sandbox is used in many system services in the Mac OS X desktop and server operating systems; however, existing applications incorporate strong assumptions of *ambient authority*, or the ability to access any object in the system (perhaps at the request of the user). With the development of the iPhone, the opportunity arose for Apple to break fundamental assumptions incorporated into existing applications. One of those assumptions was the assumption of ambient authority: in the iPhone, iPod Touch, and iPad, applications execute with a fundamental assumption of isolation from system services and other applications. Above all, the goal of this approach is to ensure robustness and reliable recovery: application programmer errors are prevented from causing problems with the device's critical services (such as being a phone), or with other applications. As such, almost all applications, including Apple's own applications, are sandboxed in iOS.

4.3.9 Performance optimisations

As part of my PhD research, I put significant effort into improving MAC Framework performance to the point where the framework could be enabled in the default FreeBSD kernel. However, the version of the MAC Framework shipped in Mac OS X and iOS was branched (and even shipped in Apple products) before many of those optimisations have been introduced. In order to meet its own performance goals for security, as well as address the specific tradeoffs present in its products, Apple produced its own set of performance optimisations, with a goal of minimising overhead for applications. Some

of these optimisations were similar to, or derived from, FreeBSD optimisations; others were significantly different.

Conditional configuration of labelling and some enforcement

In the version of the MAC Framework shipped with Mac OS X, a number of labelling and access control features required for ubiquitously labelled and enforced policies such as Biba, MLS, and TE are not enabled by default. For example, while labelling and control of Mach tasks and ports is supported in the framework, it is not compiled into the default Mac OS X kernel; similarly, support for labelling of sockets and network-layer access control checks is disabled in the default kernel configuration.

Although support for labelling `vnodes` is compiled into the default Mac OS X kernel, a global switch enabling label allocation for `vnodes` is disabled by default. Instead, policies can specifically request that labels be allocated for `vnodes` that specifically require it by calling `vnode_label` on the `vnode`. As policies shipped with Mac OS X do not make use of `vnode` labelling, this avoids a significant memory overhead, while still allowing policies that might require `vnode` labels to use them.

Conditional enforcement by thread

In FreeBSD, optimisations on entering the MAC Framework rely to a large extent on an all-or-nothing distinction between sites willing to pay the synchronisation overhead of the framework, vs. sites that are not. This is consistent with the design of mandatory policies ranging from MLS to `mac_bsdextended`, in which the goal is to apply policy to all processes in the system. In Mac OS X, however, the assumption is that sandboxing will apply only to specific high-risk processes, making it desirable to avoid the overhead of enforcement on other processes.

To this end, each process carries a flags field, `p_mac_enforce`, indicating which object types require protection for that process. Given the micro-benchmark results in Chapter 3, it might be desirable to apply a similar approach in FreeBSD in order to avoid, for example, network performance overhead when using a file system-only access control policy such as `mac_bsdextended`. This assumption of partial enforcement becomes less true in a system like iOS, in which most or all processes will be subject to some form of sandboxing; as the Mac OS X application model evolves, and application developers become more used to the idea of sandboxing, it seems desirable that this will be the case there as well. In that case, applying further optimisations developed in FreeBSD to Mac OS X, such as the use of read-mostly locks, might also be appropriate.

4.3.10 Policy label data synchronisation requirements

Development work to introduce fine-grained locking in the FreeBSD and Mac OS X kernels took place simultaneously, but also independently. The starting code bases are similar, but minor design differences have led to significantly different synchronisation properties between the two OSs. One important difference lies in synchronisation of

file system objects in VFS: in FreeBSD, the notion of a `vnode` lock shared between the VFS and the file system implementation is maintained, and the MAC Framework is able to borrow this lock to protect its own metadata.

In Mac OS X, the significance of VFS layer locking is reduced, and as the possibility of making the MAC Framework a supported KPI is considered, masking the internal locking strategy of the operating system from third-party policy developers is increasingly desirable. As such, Mac OS X policy modules are not able to integrate as directly with the kernel synchronisation model as in FreeBSD, leading to the potential for race conditions, and also greater overhead due to policies having to separately lock label state that in FreeBSD could rely on borrowed OS locks.

4.3.11 Conclusions on Mac OS X and iOS

Adoption by Apple has been one of the greatest technology transfer successes of the MAC Framework: it has simultaneously met the critical needs of a major systems vendor in supporting the development of new access control policies, and been deployed over millions of devices ranging from smart phones to high-end server hardware. It has also proven one of the most challenging adaptions of the framework, in part due to the integration of Mach and BSD in the XNU kernel. However, as applications become more complex, and even multi-million line "monolithic" kernels such as FreeBSD's appear small in comparison to applications, similar philosophical issues raised will arise there as well.

Finally, the interchange of features between the FreeBSD and Mac OS X versions of the framework has been effective, but much remains to do. FreeBSD has not yet adopted several interesting features in the Mac OS X version, some developed during the SEDarwin project, and others as part of Apple's adoption of the framework:

- Per-process entry point masks for access control, allowing the overhead of framework synchronisation to be avoided when loaded policies are interested in enforcing for only certain classes of objects. This approach could be extended also to deal with life cycle notifications, based on existing label allocation optimisations in FreeBSD.

- File descriptor labelling and controls, desirable to fully support FLASK/TE on FreeBSD.

- Label name registration, allowing userspace applications to query the set of available names, as well as to detect conflicts between independently authored policies attempting to use the same names for label elements.

Similarly, Mac OS X is missing several more recent changes to the FreeBSD version of the MAC Framework, most developed during my PhD research:

- Integration with the OS privilege model, and more generally, a fine-grained treatment of privileges in the kernel.

- Several performance optimisations in FreeBSD, especially automatic analysis of policies to determine labelling requirements, and SMP optimisations for framework synchronisation.
- DTrace instrumentation of the MAC Framework has proven invaluable in policy debugging, performance analysis, and most recently, testing of the MAC Framework.

4.4 Evaluation

Chapter 3 evaluated the MAC Framework using a number of more traditional systems research criteria: performance overhead measured through benchmarks, expressivity through measured with respect to a range of access control models, and specific security objectives such as avoidance of the classes of vulnerabilities inherent to system call wrappers discussed in Chapter 2. However, in light of widespread industry adoption described in this chapter, there is also the opportunity to evaluate the framework from a more pragmatic perspective: did the hypothesis of security extensibility prove correct in practice? Regardless of laboratory analyses and measurements, was the framework adequately extensible, expressive, performant, usable, and secure in production?

This section explores those questions through the experiences gained in transferring and deploying the MAC Framework in dozens of products and millions of shipped devices. This discussion is necessarily qualitative, and at times anecdotal, in nature, as it considers the results of an experiment in systems design that can have no control. However, the results contribute significantly to our understanding of the use of security in real-world products, and the impact a framework for security extensibility has adapting systems to rapidly evolving requirements.

4.4.1 The hypothesis of security extensibility

This dissertation proposes the hypothesis that "designing in" security extensibility for operating systems offers significant advantages in the presence of changing access control requirements. The MAC Framework embodies that hypothesis by offering OS vendors, third-party product vendors, and even certain end users the ability to augment the operating system access control policy. Chapter 3 asserted several specific advantages: a reference monitor structure with assurance benefits, reduced cost when OS vendors wish to offer extended security policies based on commodity operating systems, that third-party security extensions would be better-supported, and that access control research and technology transfer would be facilitated.

The MAC Framework has proven fertile ground in confirming all of these claims: security vendors have used the framework to develop and integrate a broad range of security policies documented in this chapter, but also as a means of improving security arguments as seen in Wu et al's static analysis work and high-assurance evaluation of

McAfee's Sidewinder firewall [150, 151, 79]. McAfee's and nCircle's experiences have similarly confirmed that a modular security framework facilitates long-term maintenance of third-party security extensions against an upstream vendor operating system making significant implementation changes over time [79, 93]. Likewise, the MAC Framework has provided infrastructure for research and development of new access control models in FreeBSD and Mac OS X, including Apple's Sandbox [1] – a security model markedly different from the labelled policies for which the framework was designed, yet well-expressed using the MAC Framework.

An evolving framework

Experience in industry has confirmed that the MAC Framework's imposition of structure on security extension assists developers in the formulating and representing security policies, and reduces the maintenance cost of those policies over the long term. However, this chapter has also documented non-trivial changes to the MAC Framework that were required to meet the needs of consumers:

- Significant performance improvement and functional refinement was required in order to enable the MAC Framework in the default FreeBSD kernel, as well as greater attention to kernel programming and binary interface compatibility. These had little effect on the programming model for policy writers, but did require reimplementation of many elements of the framework itself.
- As the FreeBSD operating system evolved, new kernel features required the introduction of new MAC Framework entry points; in general, these additions followed the model of introducing new objects and/or methods as OS services expanded. Examples of these changes include the addition of new POSIX shared memory and semaphore objects and methods.
- The addition of DTrace to FreeBSD allowed integrated tracing facilities to be added to the MAC Framework, improving the process of monitoring and debugging of policies and their compositions during development and in production.
- In some cases, it was necessary to extend the framework in order to support desirable policies – primarily, to allow policies to instrument the kernel privilege model, which required additional entry points and the new ability to grant, rather than restrict, rights. The change normalised access control for many system administration functions by making privilege checks effectively act as methods on a singleton system object. This approach appeared, but was not handled consistently, in the original design, being limited to a few key system events such as `reboot`; this enhancement improved existing policies as well, such as the Biba integrity policy, which is as a result able to usefully constraint the root user.

An important question, then, is whether the need to modify the framework in order to express new policies invalidates the claim that an extensibility framework

limits the need for operating system changes when developing access control policies. I argue that the evolution of the framework, even with respect to privileges, maintains its original central design choices: policy modules augment the OS security policy, implement controls on objects and methods of kernel-backed objects, are composed using a simple meta-policy, and build on common policy infrastructure such as object labelling and policy-agnostic management APIs. The addition of privilege constraints arguably reflects an oversight in the original implementation, which omitted control on a critical implied (rather than explicit) kernel object: the operating system itself. In all senses, it seems that that the framework has accomplished (and exceeded) its original goals of access control extensibility, and that changes to the framework over its lifetime have reflected incremental, rather than transformative modifications to its approach.

4.4.2 Expressiveness

Expressiveness is a central concern in the MAC Framework design, both from the perspective of what is made easy to express, and what is not. The choice to create a large number of reference policies, described in Chapter 3 was motivated by a desire to identify common requirements across policies and ensure that a broad range of policies could be easily and efficiently expressed using the framework. A related goal was to discourage the expression of policies that are difficult or impossible to implement correctly – in stark contrast to system call interposition, which is extremely expressive but encourages the implementation of difficult-to-implement security policies (i.e., ones that necessarily suffer security vulnerabilities as documented in Chapter 2). These factors led to a subject-object-method design, reflecting the essentially object-oriented structure of the kernel and its approach to synchronisation, but also the needs of security policies, which often centre around the flow of information between subjects and objects, interpreted through methods.

During the design and implementation of the framework, two challenges to expressiveness were identified: the necessity for revocation and floating labels (flip sides of the same coin), and the desire to express file system policies with respect to paths, which I will now consider in greater detail.

Revocation and floating labels

One of the key attributes of most security policies is their ability to express security configuration changes – typically this occurs in one of two forms: the update of a global rule list (such as in `ugidfw`), or via changes to labels on subjects or objects (such as in `mac_biba` or `mac_mls`). When policy configuration changes, the rights of subjects may expand or contract; in the latter case, this raises the question of what happens to any existing "open" references by subjects to objects to which they should no longer have access: rights could be retained (leading to a period of inconsistency) or revoked.

Different policies have different requirements: the `ugidfw` policy tracks the semantics of UNIX open files, which perform an access control check at time of `open`, and if file

ownership or permissions change at a later time, I/O access continues. Other policies may demand stronger consistency: if MLS is to prevent the flow of information in violation of its confidentiality constraints, then relabelling of an object should lead to revocation of held rights in order to avoid inappropriate read up or write down operations.

Many kernel operations are authorised as access begins, and given kernel locking, these checks are atomic with the access itself – label operations are suspended until outstanding operations complete. However, other operations are long-lived (potentially beyond the lifetime of a system call) – especially file `open` and memory mapping through `mmap`. For `open`, revocation can be implemented (if required) by policies during later I/O calls, leaving the descriptor present but limited to only uses permissible after a relabel operation; this approach is taken by `mac_biba` and `mac_mls`, which offer a global configuration flag to enable file descriptor revocation (disabled by default).

Memory mapped I/O presents a significant challenge, however: no further system calls will be issued once a page from the buffer cache is mapped directly into process memory; if revocation is required, then the VM data structures must be walked to identify and invalidate existing references (a significant book-keeping problem), and leading to difficult-to-handle application faults as existing mappings become invalid. The MAC Framework, as a result of this complexity, implements revocation support for memory mappings following a subject relabel operation, but not an object relabel operation, instead requiring new objects to be created rather than relabelled.

This approach is sufficient to implement support for LOMAC, which downgrades subjects based on a data taint model, but provides weak support for object relabelling combined with memory mapping. In order to limit application compatibility problems, modifications to process memory mappings on downgrade also make use of copy-on-write, rather than true revocation, to divorce data "before" and "after" the subject relabel: this is important, as it is desirable for copy-on-write versions of pages (such as linked binary pages) to remain functional, even though they might notionally be high integrity. In practice, these design choices have proven adequate for many prior implementations of Biba and MLS, but represent an edge case in the subject-object-method access control model, which assumes explicit, rather than implicit, operations.

A further concern with relabelling occurs because of integration between the kernel locking model and the MAC Framework: when an object label is protected by a shared lock on entry to an entry point, mutation of the label is not permitted by the synchronisation model. Labelled policies such as Biba and MLS modify subject and object labels only in MAC Framework relabel system calls where exclusive locks are held consistently. However, floating label policies such as LOMAC or DIFC[13] may mu-

[13]Despite this challenge, it appears that DIFC may well be easily implementable using the MAC Framework, as it takes a very similar perspective on subject-object-method and information flow as policies such as Biba and LOMAC. This is facilitated by the fact that policy modules can determine their own policies for label interpretation and operations – in this sense as well, the MAC Framework

tate labels at many more points in kernel operation. If a policy module needs to float a label on a subject or object at an arbitrary point in time, it may need to supplement framework-provided synchronisation with its own label data locks. In the case of subject labels, the copy-on-write model used for process credentials makes this particularly tricky: subject label changes require memory allocation, an operation that cannot be performed reliably in arbitrary kernel paths. For the purposes of LOMAC, which may relabel a subject while in the file system or network code following exposure to tainted data, the subject relabel operation must be deferred to another context – typically the next system call return (ensuring that the subject has been relabelled before the operation returns to userspace).

Access control and pathnames

One of the most common queries received about authoring MAC Framework policies on the FreeBSD operating system has to do with the enforcement of access control properties based on the path of the file being accessed. On Mac OS X, path-based controls are a central part of the Sandbox access control policy, which allows files to be selected for control using regular expressions on file system paths.

On FreeBSD, however, providing that facility is extremely difficult, due to its weak notion of file system path: a file can have zero names, one name, or multiple names due to unlinked files and hard links; worse, the same file may have different names for different processes or the same name might refer to different files (due to `chroot`) and mount point overlays). This expressiveness constraint is a serious limitation to the MAC Framework, but also not one that the framework itself can address – solving this problem requires changing the semantics of file system paths in the FreeBSD VFS, perhaps along the lines of similar changes in Mac OS X. This problem has also arisen in the context of security event auditing, in which users desire (and expect) to audit records to identify objects by name, but the system is unable to reliably meet this need in important edge cases [144].

4.4.3 Complexity

The MAC Framework has roughly 240 MAC policy entry points available for policy modules to implement, and instruments the operation of dozens of core kernel data types; the implementation of the MAC Framework itself is several thousand lines of C code. Given this apparent complexity, it is desirable to consider whether the framework could accomplish its goals with a less complex formulation.

One answer to this question is that the number of MAC Framework interfaces corresponds to the product of the number of core kernel objects requiring controls, and the number of methods on those objects – i.e., that the complexity of the framework corresponds directly to core complexity in the kernel itself, and that reducing that exposed complexity would reduce necessary expressiveness. Another perspective is

is able to implement policies that resemble discretionary, not just mandatory, access control.

that the level of complexity of the framework itself is quite small compared to the overall kernel: a few thousand lines of C code allow instrumentation of all the core security functions of the kernel, and that the only 3600 lines of C code implementing the `mac_biba` module is in fact extremely small for the implementation of a ubiquitous information flow policy spanning all the major communication channels of a UNIX operating system.

A reasonable conclusion is that although the specific presentation of the complexity might differ, that the current level of complexity is minimised. As mentioned in Chapter 3, one design choice explored early in development of the framework was whether to take the approach of using separate entry point function pointers for different object-method pairs, or to use a smaller number of entry points with arguments indicating what object type of method was involved – an approach adopted in Kauth [9]. In transitioning to explicit methods, access to the C language's type system was improved, and numerous bugs were discovered and corrected. This suggests that an improvement in programming language to facilitate flexibility while maintaining typing (such as an explicit object system) might allow for an improvement in presentation of necessary complexity.

4.4.4 Usability

Security usability is an evolving area of research in which there are few absolutes and many challenges. In the context of the MAC Framework, the are two important usability considerations: programmability for security policy authors, and any impact the MAC Framework has on user experience.

With respect to programmability, the critical question is whether policy writers are able to easily and successfully implement policy modules that capture their intents. The experiences of companies described in this chapter is that the MAC Framework eases the task of policy writing considerably: little if any kernel code must be modified (also reducing the cost of code maintenance), and the quantity of code required to implement a policy is reduced as the framework provides common infrastructure such as object labelling, system calls, and command line tools. Writing a new kernel security model still requires significant knowledge of operating systems – a requirement I believe cannot be eliminated with current systems. While these conclusions are qualitative, the widespread adoption and use of the framework for these purposes reflects it success in this regard.

Usability from a user perspective is a challenging issue, and one that the MAC Framework has relatively little influence over – it is typically the management requirements and semantics of security policies that affect usability, not the kernel instrumentation framework by which they accomplish their goals. However, at least in the area of UNIX command line tools, allowing a small set of policy-agnostic command line tools, rather than policy-specific tools, likely helps usability. Anecdotally, it appears that useful policies implemented with the framework can improve system security without

harming usability: Mac OS X and iOS have been lauded for their usability properties, and make extensive use of operating system access control.

4.4.5 Performance

In Chapter 3, I considered the performance impact of the MAC Framework through benchmarks, deriving frequently found security and performance results: a small but measurable overhead on operation. In general, the performance goal evolved through interactions with the open source community and with commercial consumers was to experience an overhead of 3% or less for enforced security policies in embedded devices and appliances. This goal was met, excepting network performance with ubiquitously labelled MAC policies, following an extended period of optimisation and real-world use. Specific efforts were made to improve performance for networking and the file system, many details of which are considered in this and the previous chapter.

The performance optimisation path taken for the FreeBSD operating system reflects a desire to minimise performance overhead for the case where `options MAC` (compiling in the framework) is minimised, and to minimise the overhead of hardening policies. This is accomplished by making many of the costs of labelled policies conditional on those features being used – for example, network packet label storage is not allocated unless a policy specifically requires it. In contrast, the focus of performance optimisation in Mac OS X was to permit additional overhead for policy enforcement – but only for processes that were explicitly "sandboxed." This led to a strategy of optimising enabling/disabling access control checks on a per-thread basis.

In general, it is possible to assert that the performance goal of minimising the cost of the framework was met in the most pragmatic sense, by virtue of the framework being accepted for use across a broad range of products.

4.4.6 Security

Metrics are a challenging issue in security research and security product development – effective metrics for "system security" remain elusive. From the perspective of the MAC Framework, three questions appear of immediate import:

- Has the structure of the framework proven resistant to vulnerability?
- Has the structure of the framework deterred vulnerabilities in policies implemented with the framework?
- Have systems built using the framework and its policies proven resistant to vulnerability?

There are no controls available in evaluating any measures of vulnerability counts; the closest we can come is to observe that few vulnerabilities have been found through efforts with static and dynamic analysis described in this chapter, nor have any been reported in the field. By design, the framework has explicitly eliminated broad classes of

security vulnerabilities such as those present in system call interposition systems – tight integration of framework and policies to the kernel synchronisation model has prevented any known instances of time-of-check-to-time-of-use vulnerabilities that were rife and trivially exploitable in wrapper systems discussed in Chapter 2. Similarly, products based on the MAC Framework (such as McAfee's Sidewinder) appear to pass high assurance security evaluations, and be strongly resistant to attacks in the field despite a clear and present threat (Apple's iOS).

A more detailed consideration of iOS reveals considerable subtlety in this evaluation, however. Despite a lack of a comprehensive public catalogue of iOS vulnerabilities, several trends emerge, in part as a result of widespread efforts to provide "jail breaking" tools for end users unhappy with the policy constraints imposed on the iPhone and iPad, such as mobile phone network lock-in:

- Exploits against iPhoneOS 1.x applications; that version of the operating system neither supported third-party applications, nor sandboxing using the MAC Framework. As a result, any application compromise was effectively a root compromise of the phone – an illustration of the risks of not deploying sandboxing.

- "Tethered" exploits, which attack the complex and (apparently) entirely privileged software components on the iPhone used to synchronise, backup, and update iOS and its applications when plugged into a USB cable. These vulnerabilities require physical access to the device, and are the preferred mechanism for jail breaking.

- "Untethered" exploits against applications run without sandboxing, or with inadequate sandboxing, such as some of Apple's own applications. As with iPhoneOS 1.x vulnerabilities, these vulnerabilities lead to compromise of the phone; anecdotally, several appear to have been in PDF rendering. These vulnerabilities reveal under-application of sandboxing techniques.

- "Untethered" exploits against the OS kernel or services by sandboxed applications, such as a recent vulnerability in the kernel's parsing of Mach-O binary headers, yielding kernel-level access. Such vulnerabilities can be attacked by either applications directly (perhaps due to a malicious application author), or following compromise of a vulnerable application yielding arbitrary code execution on the device. Such attacks violate the kernel and process isolation assumptions described in Chapter 1, and hence are able to bypass access control.

- One widely-discussed bug in which the sandboxing policy shipped with the phone failed to protect the received SMS database from reading by arbitrary applications – an error in policy-authoring.

These categories of vulnerabilities reveal both the strengths and the weaknesses of sandboxing implemented using the MAC Framework. Vulnerabilities in arbitrary

applications yielded complete control of the phone in iPhoneOS 1.x, an effect largely eliminated in later OS versions through the introduction of sandboxing. Instead, attackers must find gaps in either the policy (such as inadequately sandboxed applications written by Apple, such as phone-side synching), or kernel vulnerabilities that violate non-bypassability requirements for MAC Framework policies. It seems clear, however, that without sandboxing, these issues would be significantly worse, especially with regard to malicious application authors.

Overall, a decade of experience with the MAC Framework, half of which involved widespread deployment of the technology, appears to offer prima facia evidence that the security goals of the framework have been accomplished to an adequate extent to meet the expectations of industry, but also to significantly improve the state of the art for operating system security. The effectiveness of the framework depends on the effectiveness of its OS foundations and the correct implementation of its policies, but the framework itself has proven a firm yet flexible foundation for significant security improvements.

4.5 Conclusion

In this chapter, I have considered the TrustedBSD MAC Framework's transition from a DARPA research project to a critical and widely deployed security component across a broad range of open source and commercial products. We have reviewed how and why the framework was selected as the means to deliver new access control policies, the nature of the policies used, and in some cases, the changes required to the framework in order to meet the specific needs of products. We have considered in particular three case studies: the open source FreeBSD operating system, nCircle's enforcement appliances, and Apple's Mac OS X and iOS operating systems. The role of open source in technology transfer success should not be underestimated. The FreeBSD operating system community proved a fertile environment in which to develop a new security technology, and a wide community of users who have returned changes in order to reduce their maintenance costs – despite a license that does not obligate that return.

While the prototype security policies developed in Chapter 3 have seen use by the FreeBSD user community, perhaps the greatest research success of the framework has been in supporting the development and maintenance of unanticipated policies in third-party products: selective sandboxing, parental controls, and download tainting in Mac OS X, sandboxing and code signing of applications on iOS, and customisation of the UNIX security policy in nCircle's policy enforcement appliance. Beyond those discussed in this chapter, the MAC Framework has seen use in McAfee's high-assurance Sidewinder firewall, which implements a custom type enforcement policy, widespread deployment in Juniper's routers as part of the JunOS SDK, where the MAC Framework similarly provides isolation and robustness for third-party components, and in services such as Seccuris's monitoring service, which employs MAC-enforced isolation

of components exposed to untrustworthy data.

Finally, the use of the MAC Framework as a building block in industry, a few examples of which have been highlighted in this chapter, has led to significant impact on real-world systems, improving security and reliability of critical network infrastructure, network services, and consumer devices. I have considered the framework's effectiveness in terms of both traditional research metrics and pragmatic experiences with field deployment, evaluating the hypothesis of security extensibility with respect to expressiveness, complexity, usability, performance, and security.

Chapter 5

Capsicum: practical capabilities for UNIX

The TrustedBSD MAC Framework has proven a powerful tool in support of operating system access control extensibility, seeing ready and successful deployment in a host of embedded, appliance, and Internet service provider environments. In this chapter I consider Capsicum, a new application-centric security model blending the UNIX security model with the neglected capability security paradigm in order to support operating system access control extensibility for security-aware applications.

The MAC Framework approach is fundamentally system-centric: system policies are encapsulated in kernel modules, and managed by administrators, system integrators, and device vendors. The transition from multi-user computers to multi-computer users has not invalidated its approach: system-centric security remains critical to protect the TCB and impose global constraints on operation. A purely system-centric view, however, fails to address the observation that the security interactions of "users" are decreasingly central: desktop and notebook computers, tablet PCs, and smart phones typically have exactly one user.

Instead, applications themselves represent the competing interests of many different parties: the user (perhaps the owner), the application writer, the authors of software components (such as plugins) to the application, not to mention the multiple authors of (potentially active) content it may provide access to via web integration. While the MAC Framework performs well in containing not just users at ISPs, but also third-party applications in the iPhone, and third-party components on Juniper routers, it provides little infrastructure and no philosophy for supporting applications that themselves implement security models.

Even a single application can have complex security requirements – web browsers, for example, are execution environments for mobile (and potentially malicious) code originating from mutually untrusting sites, and employ security techniques such as Java sand-boxing to isolate them. The potential impact of such vulnerabilities is significant: the web browser simultaneously hosts diverse code ranging from JavaScript cloud-based

mail clients (such as Google Mail) to Flash applications (such as YouTube), while also handling sensitive content from medical, banking, and e-commerce web sites. Failure of separation could allow sites to eavesdrop on each others' content, or directly manipulate user input, displayed content, or downloaded files. Further, malicious code might gain access to the ambient authority of the user beyond the web browser context, accessing local file data from other applications, system configuration information, key storage, or hardware devices such as radios and cameras.

Mapping distributed security models into local enforcement mechanisms is a key aspect to security extensibility, and applications are the point of confluence between those two worlds. Current OS security facilities solve this security problem poorly due to their system-centric focus and requirement for privilege originating from the assumption that the only relevant security state transition is between users. While a MAC Framework policy could be turned to this end, it is worth observing a fundamental limit on the framework: it is intended to restrict, rather than facilitate, processes by limiting rather than creating opportunities for communication and sharing via the operating system. Capsicum relies on enhancing certain OS APIs, such as process IDs, for capability operation, allow them to be delegated in a way not previously possible in UNIX.

In this chapter, I present Capsicum, an application capability security framework targeted at a new class of security-aware applications. Capsicum extends POSIX to add new facilities, including *capability mode* and *capabilities*, which augment traditional OS facilities by providing sandbox and granular delegation APIs. Capsicum, and the capability model it depends on, are fundamentally enabling technologies, structured around the constructive act of security delegation, requiring a separate integration with the operating system. The problems addressed by the MAC Framework and Capsicum are not strictly orthogonal, but the composition of the two approaches successfully captures notions of system-centric and application-centric controls.

Capsicum was first described in a paper presented at the 2010 USENIX Security Symposium in Washington, DC. This chapter is based on that paper, but includes additional material on static analysis of concurrency properties presented at the 4th Analysis of Security APIs (ASA-4) workshop in Edinburgh, Scotland.

5.1 Introduction

Capsicum is an API that brings capabilities to UNIX. Capabilities are unforgeable tokens of authority, and have long been the province of research operating systems such as PSOS [96] and EROS [124]. UNIX systems have less fine-grained access control than capability systems, but are very widely deployed. By adding capability primitives to standard UNIX APIs, Capsicum gives application authors a realistic adoption path for one of the ideals of OS security: least-privilege operation. We validate our approach through an open source prototype of Capsicum built on (and now planned for inclusion

in) FreeBSD 9.

Today, many popular security-critical applications have been decomposed into parts with different privilege requirements, in order to limit the impact of a single vulnerability by exposing only limited privileges to more risky code. Privilege separation [103], or *compartmentalisation*, is a pattern that has been adopted for applications such as OpenSSH, Apple's SecurityServer, and, more recently, Google's Chromium web browser. Compartmentalisation is enforced using various access control techniques, but only with significant programmer effort and significant technical limitations: current OS facilities are simply not designed for this purpose.

The access control systems in conventional (non-capability-oriented) operating systems are *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC). DAC was designed to protect users from each other: the owner of an object (such as a file) can specify *permissions* for it, which are checked by the OS when the object is accessed. MAC was designed to enforce system policies: system administrators specify policies (e.g. "users cleared to Secret may not read Top Secret documents"), which are checked via run-time hooks inserted into many places in the operating system's kernel.

Neither of these systems was designed to address the case of a single application processing many types of information on behalf of one user. For instance, a modern web browser must parse HTML, scripting languages, images and video from many untrusted sources, but because it acts with the full power of the user, has access to all his or her resources (such implicit access is known as ambient authority).

These mechanisms vary by platform, but all require a significant amount of programmer effort (from hundreds of lines of code or policy to, in one case, 22,000 lines of C++) and, sometimes, elevated privilege to bootstrap them. Our analysis shows significant vulnerabilities in all of these sandbox models due to inherent flaws or incorrect use (see Section 5.5).

Capsicum addresses these problems by introducing new (and complementary) security primitives to support compartmentalisation: *capability mode* and *capabilities*. Capsicum capabilities should not be confused with operating system privileges, occasionally referred to as capabilities in the OS literature. Capsicum capabilities are an extension of UNIX file descriptors, and reflect rights on specific objects, such as files or sockets. Capabilities may be delegated from process to process in a granular way in the same manner as other file descriptor types: via inheritance or message-passing. Operating system privilege, on the other hand, refers to exemption from access control or integrity properties granted to processes (perhaps assigned via a role system), such as the right to override DAC permissions or load kernel modules. A fine-grained privilege policy supplements, but does not replace, a capability system such as Capsicum. Likewise, DAC and MAC can be valuable components of a system security policy, but are inadequate in addressing the goal of application privilege separation.

We have modified several applications, including base FreeBSD utilities and Chromium, to use Capsicum primitives. No special privilege is required, and code changes are min-

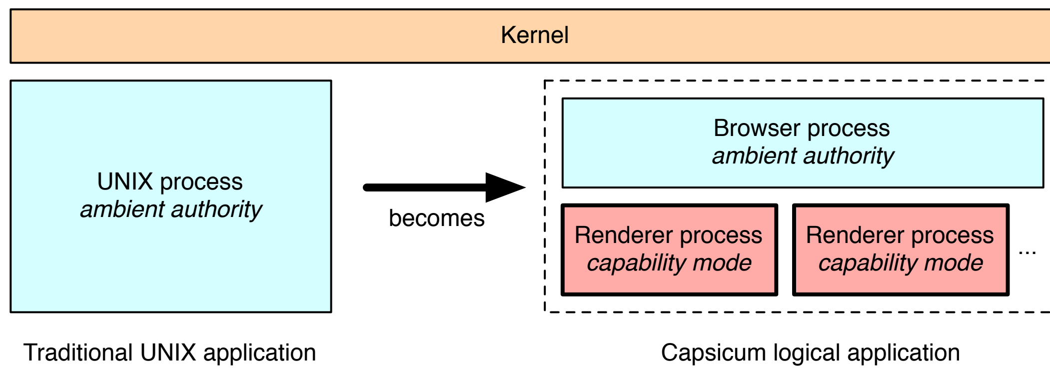

Figure 5.1: Capsicum helps applications self-compartmentalise.

imal: the `tcpdump` utility, plagued with security vulnerabilities in the past, can be sandboxed with Capsicum in around ten lines of code, and Chromium can have OS-supported sandboxing in just 100 lines.

In addition to being more secure and easier to use than other sandboxing techniques, Capsicum performs well: unlike pure capability systems where system calls necessarily employ message passing, Capsicum's capability-aware system calls are just a few percent slower than their UNIX counterparts, and the `gzip` utility incurs a constant-time penalty of 2.4 ms for the security of a Capsicum sandbox (see Section 5.6).

5.2 Capsicum design

Capsicum is designed to blend capabilities with UNIX. This approach achieves many of the benefits of least-privilege operation, while preserving existing UNIX APIs and performance, and presents application authors with an adoption path for capability-oriented design.

In order to protect user data from malicious JavaScript, Flash, etc., the Chromium web browser is decomposed into several OS processes. Some of these processes handle content from untrusted sources, but their access to user data is restricted using DAC or MAC mechanism (the process is *sandboxed*).

Capsicum extends, rather than replaces, standard UNIX APIs by adding kernel-level primitives (a sandboxed *capability mode*, *capabilities* and others) and userspace support code (*libcapsicum* and a *capability-aware run-time linker*). Together, these extensions support application *compartmentalisation*, the decomposition of monolithic application code into components that will run in independent sandboxes to form *logical applications*, as shown in Figure 5.1.

Capsicum requires application modification to exploit new security functionality, but this may be done gradually, rather than requiring a wholesale conversion to a pure capability model. Developers can select the changes that maximise positive security impact while minimising unacceptable performance costs; where Capsicum replaces

existing sandbox technology, a performance improvement may even be seen.

This model requires a number of pragmatic design choices, not least the decision to eschew microkernel architecture and migration to pure message-passing. While applications may adopt a message-passing approach, and indeed will need to do so to fully utilise the Capsicum architecture, we provide "fast paths" in the form of direct system call manipulation of kernel objects through delegated file descriptors. This allows native UNIX performance for file system I/O, network access, and other critical operations, while leaving the door open to techniques such as message-passing system calls for cases where that proves desirable.

5.2.1 Capability mode

Capability mode is a process credential flag set by a new system call, `cap_enter()`; once set, the flag is inherited by all descendent processes, and cannot be cleared. Processes in capability mode are denied access to global namespaces such as the filesystem and PID namespaces (see Figure 5.1). In addition to these namespaces, there are several system management interfaces that must be protected to maintain UNIX process isolation. These interfaces include `/dev` device nodes that allow physical memory or PCI bus access, some `ioctl()` operations on sockets, and management interfaces such as `reboot()` and `kldload()`, which loads kernel modules.

Access to system calls in capability mode is also restricted: some system calls requiring global namespace access are unavailable, while others are constrained. For instance, `sysctl()` can be used to query process-local information such as address space layout, but also to monitor a system's network connections. We have constrained `sysctl()` by explicitly marking ≈30 of 3000 parameters as permitted in capability mode; all others are denied.

The system calls which require constraints are `sysctl()`, `shm_open()`, which is permitted to create *anonymous memory objects*, but not named ones, and the `openat()` family of system calls. These calls accept a directory descriptor argument relative to which `open()`, `rename()`, etc. lookups will occur; in capability mode, they are constrained so that they can operate only on objects "under" this descriptor. For instance, if file descriptor 4 is a capability allowing access to /lib, then `openat(4, "libc.so.7")` will succeed, whereas `openat(4, "../etc/passwd")` and `openat(4, "/etc/passwd")` will not.

5.2.2 Capabilities

The most critical choice in adding capability support to a UNIX system is the relationship between capabilities and file descriptors. Some systems, such as Mach/BSD, have maintained entirely independent notions: Mac OS X provides each task with both indexed capabilities (ports) and file descriptors. Separating these concerns is logical, as Mach ports have different semantics from file descriptors; however, confusing results can arise for application developers dealing with both Mach and BSD APIs, and we

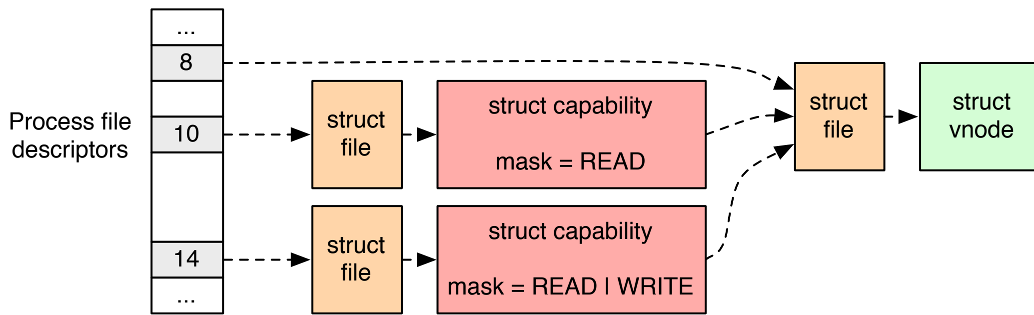

Figure 5.2: Capabilities "wrap" normal file descriptors, masking the set of permitted methods.

wanted to reuse existing APIs as much as possible. As a result, we chose to extend the file descriptor abstraction, and introduce a new file descriptor type, the capability, to wrap and protect raw file descriptors.

File descriptors already have some properties of capabilities: they are unforgeable tokens of authority, and can be inherited by a child process or passed between processes that share an inter-process communication (IPC) channel. Unlike "pure" capabilities, however, they confer very broad rights: even if a file descriptor is read-only, operations on meta-data such as `fchmod()` are permitted. In the Capsicum model, we restrict these operations by wrapping the descriptor in a capability and permitting only authorised operations via the capability, as shown in Figure 5.2.

The `cap_new()` system call creates a new capability given an existing file descriptor and a mask of rights; if the original descriptor is a capability, the requested rights must be a subset of the original rights. Capability rights are checked by `fget()`, the in-kernel code for converting file descriptor arguments to system calls into in-kernel references, giving us confidence that no paths exist to access file descriptors without capability checks. Capability file descriptors, as with most others in the system, may be inherited across `fork()` and `exec()`, as well as passed via UNIX domain sockets.

There are roughly 60 possible mask rights on each capability, striking a balance between message-passing (two rights: send and receive), and MAC systems (hundreds of access control checks). We selected rights to align with logical methods on file descriptors: system calls implementing semantically identical operations require the same rights, and some calls may require multiple rights. For example, `pread()` and `preadv()`, which read file data into memory, both require `CAP_READ` in a capability's rights mask, and `read()` (read bytes using the file offset) requires `CAP_READ | CAP_SEEK` in a capability's rights mask.

Namespace	Description
Process ID (PID)	Processes identifiers, returned by `fork()`, are used for signals, debugging, monitoring, and exit status.
File paths	Files exist in a global, hierarchical namespace, which is protected by DAC and MAC.
NFS file handles	The NFS identifies files and directories on the wire using a flat, global file handle namespace. They are also exposed to processes to support the lock manager daemon and optimise local file access.
File system ID	File system IDs supplement paths to mount points, and are used for forceable unmount when there is no valid path to the mount point.
Protocol addresses	Protocol families use socket addresses to name local and foreign endpoints. These exist in global namespaces, such as IPv4 addresses and ports, or the file system namespace for local domain sockets.
Sysctl MIB	The `sysctl()` management interface uses named and numbered entries to get or set system information, such as process lists and tuning parameters.
System V IPC	System V IPC message queues, semaphores, and shared memory segments exist in a flat, global integer namespace.
POSIX IPC	POSIX defines similar semaphore, message queue, and shared memory APIs, with an undefined namespace: on some systems, these are mapped into the file system; on others they are simply a flat global namespaces.
System clocks	UNIX systems provide multiple interfaces for querying and manipulating system clocks and timers.
Jails	The management namespace for FreeBSD-based virtualised environments.
CPU sets	Global namespace process and thread affinities.

Table 5.1: Global namespaces in the FreeBSD operating kernel

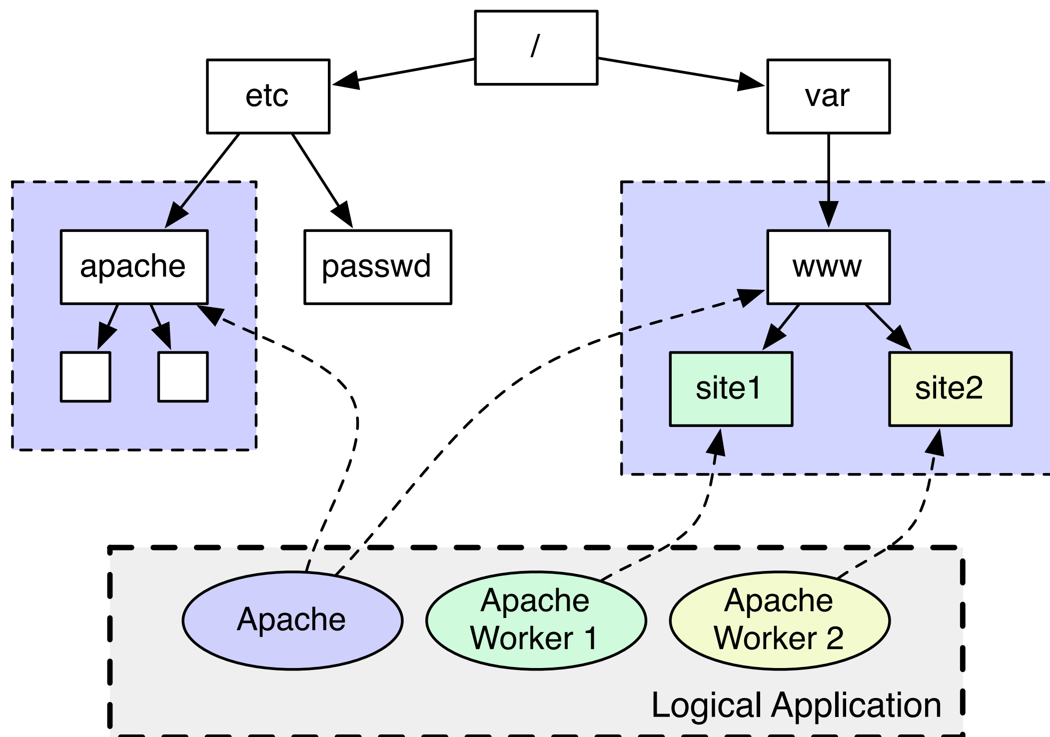

Figure 5.3: Portions of the filesystem namespace can be delegated to sandboxed processes.

Capabilities can wrap any type of file descriptor including directories, which can then be passed as arguments to `openat()` and related system calls. The `*at()` system calls begin relative lookups for file operations with the directory descriptor; we disallow some cases when a capability is passed: absolute paths, paths containing ".." components, and `AT_FDCWD`, which requests a lookup relative to the current working directory. With these constraints, directory capabilities delegate file system namespace subsets, as shown in Figure 5.3. This allows sandboxed processes to access multiple files in a directory (such as the library path) without the performance overhead or complexity of proxying each file `open()` via IPC to a process with ambient authority.

The ".." restriction is a conservative design, and prevents a subtle problem similar to historic `chroot()` vulnerabilities. A single directory capability that enforces containment by preventing ".." lookup only on the root of a subtree is correct in isolation; however, two colluding sandboxes (or a single sandbox with two capabilities) can race to rearrange a tree so that the check always passes, allowing escape from a subset. It is possible to imagine less conservative solutions, such as preventing upward renames that could introduce exploitable cycles, or additional synchronisation; these strike us as risky tactics, and we have selected the simplest solution, at some cost to flexibility.

Many past security extensions have composed poorly with UNIX security leading to vulnerabilities; thus, we disallow privilege elevation via `fexecve()` using setuid and setgid binaries in capability mode. This restriction does not prevent setuid binaries from using sandboxes.

5.2.3 Run-time environment

Even with Capsicum's kernel primitives, creating sandboxes without leaking undesired resources via file descriptors, memory mappings, or memory contents is difficult. `libcapsicum` therefore provides an API for starting scrubbed sandbox processes, and explicit delegation APIs to assign rights to sandboxes. `libcapsicum` cuts off the sandbox's access to global namespaces via `cap_enter()`, but also closes file descriptors not positively identified for delegation, and flushes the address space via `fexecve()`. Sandbox creation returns a UNIX domain socket that applications can use for inter-process communication (IPC) between host and sandbox; it can also be used to grant additional rights as the sandbox runs.

5.3 Capsicum implementation

5.3.1 Kernel changes

Many system call and capability constraints are applied at the point of implementation of kernel services, rather than by simply filtering system calls. The advantage of this approach is that a single constraint, such as the blocking of access to the global file system namespace, can be implemented in one place, `namei()`, which is responsible for processing all path lookups. For example, one might not have expected the `fexecve()` call to cause global namespace access, since it takes a file descriptor as its argument rather than a path for the binary to execute. However, the file passed by file descriptor specifies its run-time linker via a path embedded in the binary, which the kernel will then open and execute.

Similarly, capability rights are checked by the kernel function `fget()`, which converts a numeric descriptor into a `struct file` reference. We have added a new `rights` argument, allowing callers to declare what capability rights are required to perform the current operation. If the file descriptor is a raw UNIX descriptor, or wrapped by a capability with sufficient rights, the operation succeeds. Otherwise, `ENOTCAPABLE` is returned. Changing the signature of `fget()` allows us to use the compiler to detect missed code paths, providing greater assurance that all cases have been handled.

One less trivial global namespace to handle is the process ID (PID) namespace, which is used for process creation, signalling, debugging and exit status, critical operations for a logical application. A related problem for logical applications is that libraries cannot create and manage worker processes without interfering with process management in the application itself – unexpected `SIGCHLD` signals are delivered to the application, and unexpected process IDs are returned by `wait()`.

Process descriptors address these problems in a manner similar to Mach task ports: creating a process with `pdfork()` returns a file descriptor suitable for process management tasks, such as monitoring for exit via `poll()`. When the process descriptor is closed, the process is terminated, providing a user experience consistent with that of

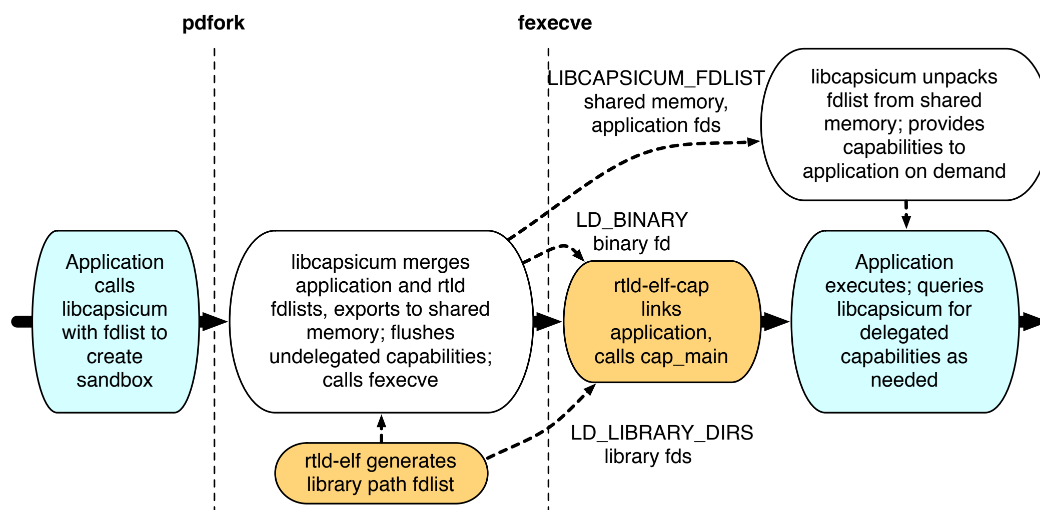

Figure 5.4: Process and components involved in creating a new `libcapsicum` sandbox

monolithic processes: when a user hits Ctrl-C, or the application segmentation faults, all processes in the logical application terminate. Termination does not occur if reference cycles exist among processes, suggesting the need for a new "logical application" primitive – see Section 5.7.

5.3.2 The Capsicum run-time environment

Removing access to global namespaces forces fundamental changes to the UNIX run-time environment. Even the most basic UNIX operations for starting processes and running programs have been eliminated: `fork()` and `exec()` both rely on global namespaces. Responsibility for launching a sandbox is shared. `libcapsicum` is invoked by the application, and responsible for forking a new process using `pdfork()`, gathering together delegated capabilities from the application and libraries, and directly executing the run-time linker, passing the sandbox binary via a capability. ELF[1] and the headers normally contain a hard-coded path to the run-time linker to be used with the binary. We execute the Capsicum-aware run-time linker directly, eliminating this dependency on the global file system namespace.

Once `rtld-elf-cap` is executing in the new process, it loads and links the binary using libraries loaded via library directory capabilities set up by `libcapsicum`. The `main()` function of a program can call `lcs_get()` to determine whether it is in a sandbox, retrieve sandbox state, query creation-time delegated capabilities, and retrieve an IPC handle so that it can process RPCs and receive run-time delegated capabilities. This allows a single binary to execute both inside and outside of a sandbox, diverging behaviour based on its execution environment. This process is illustrated in greater detail in Figure 5.4.

[1]ELF, the Executable and Linkable Format, is the file format used in most contemporary UNIX systems, with the exception of Mac OS X, which uses the Mach-O format.

Once in execution, the application is linked against normal C libraries and has access to much of the traditional C run-time, subject to the availability of system calls that the run-time depends on. An IPC channel, in the form of a UNIX domain socket, is set up automatically by `libcapsicum` to carry RPCs and capabilities delegated after the sandbox starts. Capsicum does not enforce the use of a specific Interface Description Language (IDL), as existing compartmentalised or privilege-separated applications have their own, often hand-coded, RPC marshalling already. Here, our design choice differs from historic microkernel systems, which universally have selected a specific IDL, such as the Mach Interface Generator (MIG) on Mach.

`libcapsicum`'s `fdlist` (file descriptor list) abstraction allows complex, layered applications to declare capabilities to be passed into sandboxes, in effect providing a sandbox template mechanism. This avoids encoding specific file descriptor numbers into the ABI between applications and their sandbox components, a technique used in Chromium that we felt was likely to lead to programming errors. Of particular concern is the hard-coding of file descriptor numbers for specific purposes, when those descriptor numbers may already have been used by other layers of the system. Instead, application and library components declare process-local names bound to file descriptor numbers before creating the sandbox; matching components in the sandbox can then query those names to retrieve (possibly renumbered) file descriptors.

5.3.3 Concurrency concerns with directory delegation

In earlier chapters, we have explored the issue of correctness in the presence of concurrency for security extension mechanisms – a critical consideration with contemporary operating system design. For example, whereas system call interposition suffered from fundamental races, the MAC Framework was crafted specifically to integrate security policies with the kernel's synchronisation approach. Concurrency is likewise a significant concern in the design and implementation of Capsicum, albeit in significantly different ways than those raised in Chapters 2 and 3. Much of the safety of Capsicum rests on atomicity properties already provided by the FreeBSD kernel, such as atomic validation of file descriptor policies during lookup. However, one area of significant complexity, and potentially risky concurrency, in Capsicum is its imposition of namespace delegation via directory descriptors.

Capsicum allows directory capabilities to be passed to sandboxes, as shown in Figure 5.5, granting access for specified operations to the directory and any children objects. In effect, this allows delegations of the form "The sandbox may open for read any object under `/tmp/sandbox`" or "The sandbox may read or write all files and directories under `/tmp/sandbox/foo`, creating a hybrid of the UNIX namespace and a pure capability model. While this functionality could be implemented using only passing of file descriptors, directory delegation offers another fast path avoiding the need for expensive interposition.

To provide this model of delegation, Capsicum only allows objects "below" the

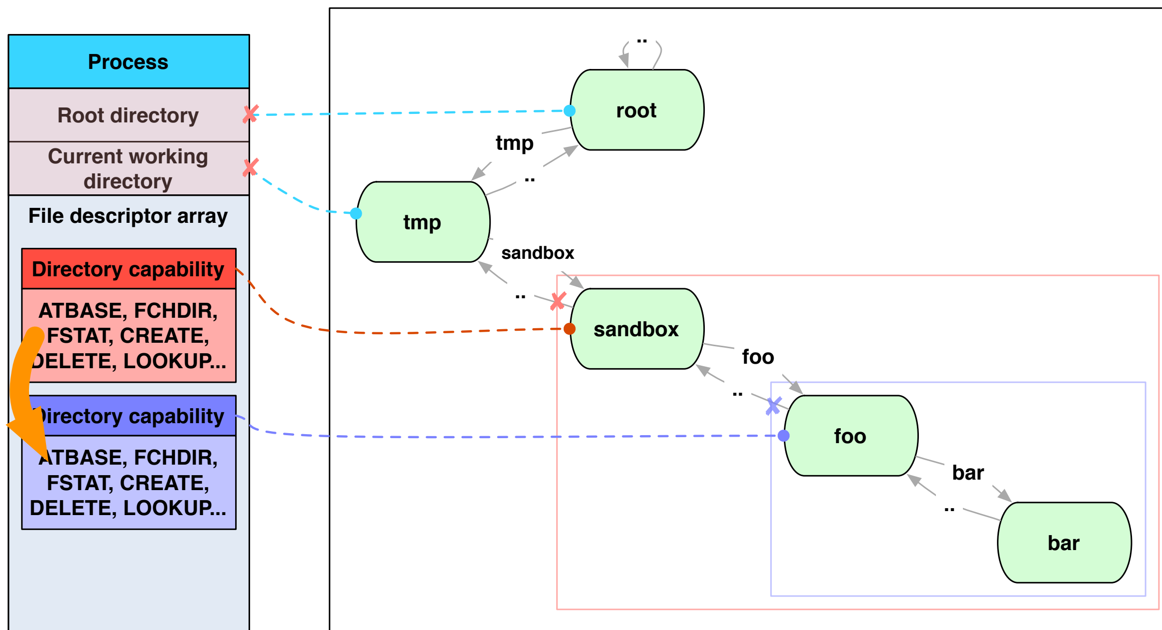

Figure 5.5: Capsicum directory delegation

directory capability to be accessed. This requires the UNIX path resolution routine, `namei()`, to implement a new invariant: during a lookup, the parent ("..") of the starting directory capability can never be named. In our initial implementation, we modified `namei()` to introduce a new constraint: any attempt to look up ".." relative to the starting directory capability would lead to access control failure. This approach has a low implementation cost, and otherwise allows full file system semantics in the subtree, such as creating, renaming, removing, and opening files and directories.

After implementing Capsicum, we encountered a concurrency vulnerability exploiting non-atomicity in `namei()`: two threads can concurrently collude in manipulating the file system to escape their respective sandboxes. Figure 5.6 illustrates how this might occur using two writable directory capabilities, one a subset of the other. When the threads simultaneously issue `openat()` and `renameat()` system calls, an in-progress `namei()` operation can experience a cycle, allowing the parent of the starting directory capability to be reached without violating the ".." lookup invariant of either directory capability. In our example, `/tmp` is reachable despite neither capability granting access to it. We were able to successfully exploit this vulnerability on a dual-core system in roughly 100,000 loops of simultaneous `openat()` and `renameat()`, illustrating that this is not simply a theoretical vulnerability.

Fundamentally, file system delegation with UNIX semantics is extremely tricky: paths are ephemeral traversal instructions, rather than first class objects. Therefore we had to consider fixes that limited UNIX semantics: our first pass disallows ".." in paths looked up using directory capabilities, preventing cycles in modifications and traversals and eliminating the vulnerability. However, this change also breaks compatibility with existing applications that might reasonably expect "..". Other semantic weakening is

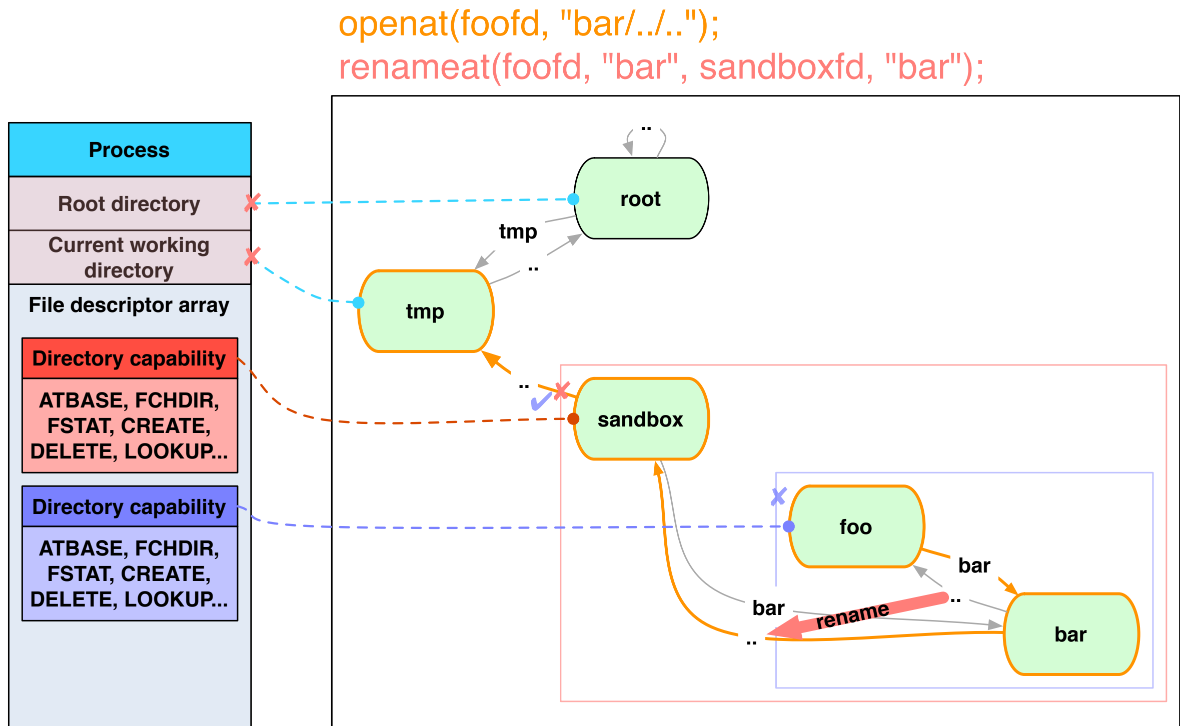

Figure 5.6: A malicious lookup which evades constraint to violate invariants

also sufficient, such as eliminating `renameat()`, and preventing concurrent namespace operations.

To better explore the problem, we employed the SPIN model checker to exhaustively test a model of the problem and potential solutions. It became clear through analysis that the fundamental problem is the creation of cycles in the namespace, a property of concurrency composed with high levels of expressiveness in the API. To this end, we have retained the restriction on looking up ".." in the final version of the Capsicum model despite its significant impact on file system semantics, as it directly addresses the presence of cycles during lookup: the file system becomes a directed acyclic graph from the perspective of colluding sandboxed processes. With this narrower model in directory capability pathname lookup, it is still possible for a sandboxed process to escape directory capability containment – but only when colluding with a process that has ambient authority (i.e., running outside of the capability system model).

5.4 Adapting applications to use Capsicum

Adapting applications for use with sandboxing is a non-trivial task, regardless of the framework, as it requires analysing programs to determine their resource dependencies, and adopting a distributed system programming style in which components must use message passing or explicit shared memory rather than relying on a common address space for communication. In Capsicum, programmers have a choice of working directly

with capability mode or using `libcapsicum` to create and manage sandboxes, and each model has its merits and costs in terms of development complexity, performance impact, and security:

1. Modify applications to use `cap_enter()` directly in order to convert an existing process with ambient privilege into a capability mode process inheriting only specific capabilities via file descriptors and virtual memory mappings. This works well for applications with a simple structure like: open all resources, then process them in an I/O loop, such as programs operating in a UNIX pipeline, or interacting with the network for the purposes of a single connection. The performance overhead will typically be extremely low, as changes consist of encapsulating broad file descriptor rights into capabilities, followed by entering capability mode. We illustrate this approach with `tcpdump`.

2. Use `cap_enter()` to reinforce the sandboxes of applications with existing privilege separation or compartmentalisation. These applications have a more complex structure, but are already aware that some access limitations are in place, so have already been designed with file descriptor passing in mind. Refining these sandboxes can significantly improve security in the event of a vulnerability, as we show for `dhclient` and Chromium; the performance and complexity impact of these changes will be low because the application already adopts a message passing approach.

3. Modify the application to use the full `libcapsicum` API, introducing new compartmentalisation or reformulating existing privilege separation. This offers significantly stronger protection, by virtue of flushing capability lists and residual memory from the host environment, but at higher development and run-time costs. Boundaries must be identified in the application such that not only is security improved (i.e., code processing risky data is isolated), but so that resulting performance is sufficiently efficient. We illustrate this technique using modifications to `gzip`.

Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing. Distributed debugging is an active area of research, but commodity tools are unsatisfying and difficult to use. While we have not attempted to extend debuggers, such as `gdb`, to better support distributed debugging, we have modified a number of FreeBSD tools to improve support for Capsicum development, and take some comfort in the generally synchronous nature of compartmentalised applications.

The FreeBSD `procstat` command inspects kernel-related state of running processes, including file descriptors, virtual memory mappings, and security credentials. In Capsicum, these resource lists become capability lists, representing the rights available to the process. We have extended `procstat` to show new Capsicum-related information,

```
+       if (cap_enter() < 0)
+               error("cap_enter: %s", pcap_strerror(errno));
        status = pcap_loop(pd, cnt, callback, pcap_userdata);
```

Figure 5.7: A two-line change adding capability mode to `tcpdump`: `cap_enter()` is called prior to the main `libpcap` (packet capture) work loop. Access to global file system, IPC, and network namespaces is restricted.

such as capability rights masks on file descriptors and a flag in process credential listings to indicate capability mode. As a result, developers can directly inspect the capabilities inherited or passed to sandboxes.

When adapting existing software to run in capability mode, identifying capability requirements can be tricky; often the best technique is to discover them through dynamic analysis, identifying missing dependencies by tracing real-world use. To this end, capability-related failures are distinguished from other failures by a new `errno` value, `ENOTCAPABLE`, and system calls such as `open()` are blocked in `namei`, rather than the system call boundary, so that paths are shown in FreeBSD's `ktrace` facility and are available to `DTrace` scripts.

Another common compartmentalised development strategy is to allow the multi-process logical application to be run as a single process for debugging purposes. `libcapsicum` provides an API to query whether sandboxing for the current application or component is enabled by policy, making it easy to enable and disable sandboxing for testing. As RPCs are generally synchronous, the thread stack in a sandbox is logically an extension of the thread stack in the host process, which makes the distributed debugging task less fraught than it otherwise might appear.

5.4.1 TCPDUMP

`tcpdump` provides an excellent example of Capsicum primitives offering immediate wins through straight-forward changes, but also the subtleties that arise when compartmentalising software not written with that goal in mind. `tcpdump` has a simple model: compile a pattern into a BPF filter, configure a BPF device as an input source, and loop, printing captured packets. This structure lends itself to sandboxing: resources are acquired early with ambient privilege, and later processing depends only on held capabilities, so can execute in capability mode. The two-line change shown in Figure 5.7 implements this conversion.

This significantly improves security, as historically fragile packet-parsing code now executes with reduced privilege. However, further analysis with the `procstat` tool is required to confirm that only desired capabilities are exposed. While there are few surprises, unconstrained access to a user's terminal connotes significant rights, such as access to key presses. A refinement, shown in Figure 5.8, prevents reading `stdin` while still allowing output. Figure 5.9 illustrates `procstat` on the resulting process, including

```
+       if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
+               error("lc_limitfd: unable to limit STDIN_FILENO");
+       if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDOUT_FILENO");
+       if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDERR_FILENO");
```

Figure 5.8: Using `lc_limitfd()`, `tcpdump` can further narrow rights delegated by inherited file descriptors, such as limiting permitted operations on `stdin` to `fstat()`.

```
  PID COMM               FD T     FLAGS CAPABILITIES PRO NAME
 1268 tcpdump             0 v rw------c           fs -   /dev/pts/0
 1268 tcpdump             1 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump             2 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump             3 v rw-------            - -   /dev/bpf
```

Figure 5.9: `procstat -fC` displays capabilities held by a process; `FLAGS` represents the file open flags, whereas `CAPABILITIES` represents the capabilities rights mask. In the case of `stdin`, only `fstat()` (`fs`) has been granted.

capabilities wrapping file descriptors in order to narrow delegated rights.

`ktrace` reveals another problem, `libc` DNS resolver code depends on file system access, but not until after `cap_enter()`, leading to denied access and lost functionality, as shown in Figure 5.10.

This illustrates a subtle problem with sandboxing: highly layered software designs often rely on on-demand initialisation, lowering or avoiding startup costs, and those initialisation points are scattered across many components. This is corrected by switching to the lightweight resolver, which sends DNS queries to a local daemon that performs actual resolution, addressing both file system and network address namespace concerns. Despite these limitations, this example of capability mode and capability APIs shows that even minor code changes can lead to dramatic security improvements, especially for a critical application with a long history of security problems. An exploited buffer overflow, for example, will no longer yield access to the file system, the ability to instantiate new connections, or the ability to load kernel modules: a significant improvement in security.

5.4.2 DHCLIENT

FreeBSD ships the OpenBSD DHCP client, which includes privilege separation support. On BSD systems, the DHCP client must run with sufficient privilege to open BPF descriptors, create raw sockets, and configure network interfaces. This creates an appealing target for attackers: network code exposed to a complex packet format while running with root privilege. The DHCP client is afforded only weak tools to constrain

```
1272 tcpdump CALL  open(0x80092477c,O_RDONLY,<unused>0x1b6)
1272 tcpdump NAMI  "/etc/resolv.conf"
1272 tcpdump RET   connect -1 errno 78 Function not implemented
1272 tcpdump CALL  socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP)
1272 tcpdump RET   socket 4
1272 tcpdump CALL  connect(0x4,0x7fffffffe080,0x10)
1272 tcpdump RET   connect -1 errno 78 Function not implemented
```

Figure 5.10: `ktrace` reveals a problem: DNS resolution depends on file system and TCP/IP namespaces after `cap_enter()`.

```
  PID COMM               FD T     FLAGS CAPABILITIES PRO NAME
18988 dhclient            0 v rw-------            - -   /dev/null
18988 dhclient            1 v rw-------            - -   /dev/null
18988 dhclient            2 v rw-------            - -   /dev/null
18988 dhclient            3 s rw-------            - UDD /var/run/log
18988 dhclient            5 s rw-------            - ?
18988 dhclient            6 p rw-------            - -   -
18988 dhclient            7 v -w-------            - -   /var/db/dhcl
18988 dhclient            8 v rw-------            - -   /dev/bpf
18988 dhclient            9 s rw-------            - IP? 0.0.0.0:0 0.
```

Figure 5.11: Capabilities held by `dhclient` before Capsicum changes: several unnecessary rights are present.

operation: it starts as the root user, opens the resources its unprivileged component will require (raw socket, BPF descriptor, lease configuration file), forks a process to continue privileged activities (such as network configuration), and then confines the parent process using `chroot()` and the `setuid()` family of system calls. Despite hardening of the BPF `ioctl()` interface to prevent reattachment to another interface or reprogramming the filter, this confinement is weak; `chroot()` limits only file system access, and switching credentials offers poor protection against weak or incorrectly configured DAC protections on the `sysctl()` and PID namespaces.

Through a similar two-line change to that in `tcpdump`, we can reinforce (or, through a larger change, replace) existing sandboxing with capability mode. This instantly denies access to the previously exposed global namespaces, while permitting continued use of held file descriptors. As there has been no explicit flush of address space, memory, or file descriptors, which is the key limitation to this approach, it is important to analyse what capabilities are available to the sandbox. Figure 5.11 shows a `procstat -fC` listing of file descriptors.

The existing `dhclient` code has done an effective job at eliminating directory access, but continues to allow the sandbox direct rights to submit arbitrary log messages to

syslogd, modify the lease database, and a raw socket on which a broad variety of operations could be performed. The last of these is of particular interest due to ioctl(); although dhclient has given up system privilege, many network socket ioctl()s are defined, allowing access to system information. The Capsicum version of dhclient blocks these potential escapes, providing more robust constraint in the event of protocol parsing bugs.

It is easy to imagine further extending the privilege separation in dhclient to use the Capsicum capability facility to further constrain file descriptors inherited in the sandbox environment, for example, by limiting the IP raw socket to send() and recv(), disallowing ioctl(). Using the libcapsicum API would require more significant changes, but as dhclient already adopts a message passing structure to communicate between its components, it would be relatively straightforward, offering better protection against capability and memory leakage. Further migration to message passing would prevent arbitrary log messages or unformatted writes to the leases file by enforcing syntax.

5.4.3 GZIP

The gzip command line tool presents an interesting target for conversion for several reasons: it implements risky compression routines that have suffered past vulnerabilities, it contains no existing compartmentalisation, and it executes with ambient user (rather than system) privileges. Historic UNIX sandboxing techniques, such as chroot() and sandbox UIDs, are a poor match because of their privilege requirement, but also because, unlike with dhclient, the notion of a single global sandbox for the application is inadequate. Many gzip sessions can run independently for many different users, and there can be no assumption that placing them in the same sandbox provides the desired security properties.

The first step is to identify natural fault lines in the application: for example, code that requires ambient privilege (due to opening files or building network connections), and code that performs more risky activities, such as parsing data and managing buffers. In gzip, this split is immediately obvious: the main run loop of the application processes command line arguments, identifying streams and files to process and send results to, and supplies input and output file descriptors to compression routines. This suggests a partitioning in which pairs of descriptors are submitted to a sandbox for processing after the ambient privilege process opens them and performs initial header handling.

We modified gzip to use libcapsicum, intercepting three core functions and optionally proxying them using RPCs to a sandbox based on policy queried from libcapsicum, as shown in Table 5.2. Each RPC passes two capabilities, for input and output, to the sandbox, as well as miscellaneous fields such as returned size, original filename, and modification time. By limiting capability rights to a combination of CAP_READ, CAP_WRITE, and CAP_SEEK, a tightly constrained sandbox is created, preventing access to any other files in the file system, or other globally named resources, in the event a vulnerability in compression code is exploited.

Function	RPC	Description
`gz_compress`	`PROXIED_GZ_COMPRESS`	zlib-based compression
`gz_uncompress`	`PROXIED_GZ_UNCOMPRESS`	zlib-based decompression
`unbzip2`	`PROXIED_UNBZIP2`	bzip2-based decompression

Table 5.2: Three `gzip` functions are proxied via RPC to the sandbox

These changes add 409 lines (about 16%) to the size of the `gzip` source code, largely to marshal and un-marshal RPCs. In adapting `gzip`, we were initially surprised to see a performance improvement; investigation of this unlikely result revealed that we had failed to propagate the compression level (a global variable) into the sandbox, leading to the incorrect algorithm selection. This serves as reminder that code not originally written for decomposition requires careful analysis. Oversights such as this one are not caught by the compiler: the variable was correctly defined in both processes, but never propagated.

Compartmentalisation of gzip raises an important design question when working with capability mode: the changes were small, but non-trivial: is there a better way to apply sandboxing to applications most frequently used in pipelines? Seaborn has suggested one possibility: a Principle of Least Authority Shell (PLASH), in which the shell runs with ambient privilege and pipeline components are placed in sandboxes by the shell [119]. We have begun to explore this approach on Capsicum, but observe that the design tension exists here as well: `gzip`'s non-pipeline mode performs a number of application-specific operations requiring ambient privilege, and logic like this may be equally (if not more) awkward if placed in the shell. On the other hand, when operating purely in a pipeline, the PLASH approach offers the possibility of near-zero application modification.

Another area we are exploring is library self-compartmentalisation. With this approach, library code sandboxes portions of itself transparently to the host application. This approach motivated a number of our design choices, especially as relates to the process model: masking `SIGCHLD` delivery to the parent when using process descriptors allows libraries to avoid disturbing application state. This approach would allow video codec libraries to sandbox portions of themselves while executing in an unmodified web browser. However, library APIs are often not crafted for sandbox-friendliness: one reason we placed separation in `gzip` rather than `libz` is that `gzip` provided internal APIs based on file descriptors, whereas `libz` provided APIs based on buffers. Forwarding capabilities offers full UNIX I/O performance, whereas the cost of performing RPCs to transfer buffers between processes scales with file size. Likewise, historic vulnerabilities in `libjpeg` have largely centred on callbacks to applications rather than existing in isolation in the library; such callback interfaces require significant changes to run in an RPC environment.

5.4.4 Chromium

Google's Chromium web browser uses a multi-process architecture similar to a Capsicum logical application to improve robustness [106]. In this model, each tab is associated with a *renderer process* that performs the risky and complex task of rendering page contents through page parsing, image rendering, and JavaScript execution. More recent work on Chromium has integrated sandboxing techniques to improve resilience to malicious attacks rather than occasional instability; this has been done in various ways on different supported operating systems, as we will discuss in detail in Section 5.5.

The FreeBSD port of Chromium did not include sandboxing, and the sandboxing facilities provided as part of the similar Linux and Mac OS X ports bear little resemblance to Capsicum. However, the existing compartmentalisation meant that several critical tasks had already been performed:

- Chromium assumes that sandboxed processes cannot open new objects
- Certain services were already forwarded to renderers, such as font loading via passed file descriptors
- Renderers transfer output to the browser via shared memory
- Separation using RPC and descriptor passing was already present

The only significant Capsicum change to the FreeBSD port of Chromium was to switch from System V shared memory (permitted in Linux sandboxes) to POSIX shared memory code as used in the Mac OS X port, which is capability-oriented and hence permitted in capability mode. Approximately 100 additional lines of code were required to introduce calls to `lc_limitfd()` to limit access to file descriptors passed to sandbox processes, such as Chromium data `pak` files, `stdio`, and `/dev/random`, font files, and to call `cap_enter()`. This compares favourably with the 4.3 million lines of code in the Chromium source tree, but would not have been possible without existing sandbox support in the design. We believe it should be possible, without a significantly larger number of lines of code, to explore using the `libcapsicum` API directly.

5.5 Comparison of sandboxing technologies

We now compare Capsicum to existing sandbox mechanisms. Chromium provides an ideal context for this comparison, as it employs six sandboxing technologies (see Table 5.3). Of these, the two are DAC-based, two MAC-based and two capability-based.

5.5.1 Windows ACLs and SIDs

On Windows, Chromium employs DAC to create sandboxes [106]. The unsuitability of inter-user protections for the intra-user context is demonstrated well: the model is both

OS	Model	LoC	Description
Windows	ACLs	22,350	Windows ACLs and SIDs
Linux	`chroot()`	605	`setuid()` root helper sandboxes renderer
Mac OS X	Sandbox	560	Path-based MAC sandbox
Linux	SELinux	200	Restricted sandbox type enforcement domain
Linux	`seccomp`	11,301	`seccomp` and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using `cap_enter()`

Table 5.3: Sandboxing mechanisms employed by Chromium.

incomplete and unwieldy. Chromium uses Access Control Lists (ACLs) and Security Identifiers (SIDs) to sandbox renderers on Windows. Chromium creates a SID with reduced privilege, which does not appear in the ACL of any object, in effect running the renderer as an anonymous user.

Objects which do not support ACLs are not protected by the sandbox. In some cases, additional precautions can be used, such as an alternate, invisible desktop to protect the user's GUI environment. However, unprotected objects include FAT filesystems on USB sticks and TCP/IP sockets: a sandbox cannot read user files directly, but it may be able to communicate with any server on the Internet or use a configured VPN. USB sticks present a significant concern, as they are frequently used for file sharing, backup, and robustness against malware.

Many legitimate system calls are also denied to the sandboxed process. These calls are forwarded by the sandbox to a trusted process responsible for filtering and serving them. This forwarding comprises most of the 22,000 lines of code in the Windows sandbox module.

5.5.2 Linux chroot

Chromium's Linux `suid()` model also attempts to create a sandbox using legacy OS access control; the result is similarly porous, but with the additional risk posed by the need for OS privilege to create the sandbox.

In this model, access to the filesystem is limited to a directory via `chroot()`: the directory becomes the sandbox's virtual root directory. Access to other namespaces, including System V shared memory (where the user's X window server can be contacted) and network access, is unconstrained, and great care must be taken to avoid leaking resources when entering the sandbox.

Initiating `chroot()` requires a `setuid()` binary: a program that runs with full system privilege. While comparable to Capsicum's capability mode in terms of intent, this model suffers from significant weakness (for example, permitting full access to the System V shared memory as well as all operations on passed file descriptors), and

comes at the cost of an additional setuid-root binary that runs with system privilege.

5.5.3 Mac OS X Sandbox

On Mac OS X, Chromium uses a MAC-based framework for creating sandboxes. This allows Chromium to create a stronger sandbox than is possible with DAC, but the rights that are granted to render processes are still very broad, and security policy must be specified separately from the code that relies on it.

The Mac OS X *Sandbox* system allows processes to be constrained according to a scheme-based policy language [53]. It uses the MAC Framework [143] to check application activities; Chromium uses three policies for different components, allowing access to filesystem elements such as font directories while restricting access to the global namespace.

As with other techniques, resources are acquired before constraints are imposed, so care must be taken to avoid leaking resources into the sandbox. Fine-grained filesystem constraints are possible, but other namespaces such as POSIX shared memory, are an all-or-nothing affair. The Seatbelt-based sandbox model is less verbose than other approaches, but like all MAC systems, security policy must be expressed separately from code. This can lead to inconsistencies and vulnerabilities.

5.5.4 SELinux

Chromium's MAC approach on Linux uses an SELinux Type Enforcement policy [78]. SELinux can be used for very fine-grained rights assignment, but in practice, broad rights are conferred because fine-grained Type Enforcement policies are difficult to write and maintain. The requirement that an administrator be involved in defining new policy and applying new types to the file system is a significant inflexibility: application policies cannot adapt dynamically, as privilege is required to reformulate policy and relabel objects.

The Fedora reference policy for Chromium creates a single SELinux dynamic domain, `chrome_sandbox_t`, which is shared by all sandboxes, risking potential interference between sandboxes. This domain is assigned broad rights, such as the ability to read all files in `/etc` and access to the terminal device. These broad policies are easier to craft than fine-grained ones, reducing the impact of the dual-coding problem, but are much less effective, allowing leakage between sandboxes and broad access to resources outside of the sandbox.

In contrast, Capsicum eliminates dual-coding by combining security policy with code in the application. This approach has benefits and drawbacks: while bugs can't arise due to inconsistencies between policy and code, there is no longer an easily accessible policy specification that can be analysed statically. This reinforces our belief that Type Enforcement and Capsicum are potentially complementary, serving differing niches in system security.

OS	Sandbox	FS	IPC	NET	S≠S'	Priv
Windows	ACLs	!	!	✗	✗	✓
Linux	`chroot()`	✓	✗	✗	✓	✗
Mac OS X	Sandbox	✓	!	✓	✓	✓
Linux	SELinux	✓	!	✓	✗	✗
Linux	`seccomp`	!	!	✓	✓	✓
FreeBSD	Capsicum	✓	✓	✓	✓	✓

Table 5.4: Comparison of security properties of Chromium sandboxes using different OS isolation models.

5.5.5 Linux seccomp

Linux has an optionally-compiled capability mode-like facility called `seccomp`. Processes in `seccomp` mode are denied access to all system calls except `read()`, `write()`, and `exit()`. At face value, this seems promising, but as OS infrastructure to support applications using `seccomp` is minimal, application writers must go to significant effort to use it.

In order to allow other system calls, Chromium constructs a process in which one thread executes in `seccomp` mode, and another "trusted" thread sharing the same address space has normal system call access. Chromium rewrites `glibc` system call vectors to forward system calls to the trusted thread, where they are filtered in order to prevent access to inappropriate shared memory objects, opening files for write, etc. However, this default policy is, itself, quite weak, as read of any file system object is permitted.

The Chromium `seccomp` sandbox contains over a thousand lines of hand-crafted assembly to set up sandboxing, implement system call forwarding, and craft a basic security policy. Such code is a risky proposition: difficult to write and maintain, with any bugs likely leading to security vulnerabilities. The Capsicum approach is similar to that of `seccomp`, but by offering a richer set of services to sandboxes, as well as more granular delegation via capabilities, it is easier to use correctly.

5.5.6 Summary of Chromium isolation models

Table 5.4 presents a comparison of the security properties of the different sandbox models. Capsicum offers the most complete isolation across various system interfaces: file system (FS), interprocess communication (IPC), and networking (NET), as well as isolating individual sandboxes from one another (S≠S'), and avoiding the requirement for OS privilege to instantiate new sandboxes (Priv). Exclamation points indicate cases where protection does exist in a model, but is either incomplete (FAT file system protection in Windows) or improperly used (file system access under the `seccomp`

sandbox is present in the underlying model, but due to poor semantics, is re-enabled by Chromium with excessive scope).

5.6 Performance evaluation

Typical operating system security benchmarking is targeted at illustrating zero or near-zero overhead in the hopes of selling general applicability of the resulting technology. Our thrust is slightly different: we know that application authors who have already begun to adopt compartmentalisation are willing to accept significant overheads for mixed security return. Our goal is therefore to accomplish comparable performance with significantly improved security.

We evaluate performance in two ways: first, a set of micro-benchmarks establishing the overhead introduced by Capsicum's capability mode and capability primitives. As we are unable to measure any noticeable performance change in our adapted UNIX applications (`tcpdump` and `dhclient`) due to the extremely low cost of entering capability mode from an existing process, we then turn our attention to the performance of our `libcapsicum`-enhanced `gzip`.

All performance measurements have been performed on an 8-core Intel Xeon E5320 system running at 1.86 GHz with 4GB of RAM, running either an unmodified FreeBSD 8-STABLE distribution synchronised to revision 201781 (2010-01-08) from the FreeBSD Subversion repository, or a synchronised 8-STABLE distribution with our capability enhancements.

5.6.1 System call performance

First, we consider system call performance through micro-benchmarking. Figure 5.12 summarises these results for various system calls on unmodified FreeBSD, and related capability operations in Capsicum. Figure 5.5 contains a table of benchmark timings. All micro-benchmarks were run by performing the target operation in a tight loop over an interval of at least 10 seconds, repeating for 10 iterations. Differences were computed using Student's t-test at 95% confidence.

Our first concern is with the performance of capability creation, as compared to raw object creation and the closest UNIX operation, `dup()`. We observe moderate, but expected, performance overheads for capability wrapping of existing file descriptors: the `cap_new()` syscall is 50.7% $\pm$ 0.08% slower than `dup()`, or 539 $\pm$ 0.8ns slower in absolute terms.

Next, we consider the overhead of capability "unwrapping", which occurs on every descriptor operation. We compare the cost of some simple operations on raw file descriptors, to the same operations on a capability-wrapped version of the same file descriptor: writing a single byte to `/dev/null`, reading a single byte from `/dev/zero`; reading 10000 bytes from `/dev/zero`; and performing an `fstat()` call on a shared memory file descriptor. In all cases we observe a small overhead of about $0.06\mu s$ when operating on the

capability-wrapped file descriptor. This has the largest relative performance impact on `fstat()` (since it does not perform I/O, simply inspecting descriptor state, it should thus experience the highest overhead of any system call which requires unwrapping). Even in this case the overhead is relatively low: $10.2\% \pm 0.5\%$.

5.6.2 Sandbox creation

Capsicum supports two ways to create a sandbox: directly invoking `cap_enter()` to convert an existing process into a sandbox, inheriting all current capability lists and memory contents, and the `libcapsicum` sandbox API, which creates a new process with a flushed capability list.

`cap_enter()` performs similarly to `chroot()`, used by many existing compartmentalised applications to restrict file system access. However, `cap_enter()` out-performs `setuid()` as it does not need to modify resource limits. Entering a capability mode sandbox is roughly twice as fast as entering a traditional UNIX `chroot()` and `setuid()` sandbox. This suggests that the overhead of adding capability mode support to an application with existing compartmentalisation will be negligible, and replacing existing sandboxing with `cap_enter()` may even marginally improve performance.

Creating a new sandbox process and replacing its address space using `execve()` is an expensive operation. Micro-benchmarks indicate that the cost of `fork()` is three orders of magnitude greater than manipulating the process credential, and adding `execve()` or even a single instance of message passing increases that cost further. The `pdfork()` system call, which returns a process descriptor from a fork operation rather than a PID, adds no measurable overhead with respect to `fork`. We also found that dynamically linked library dependencies (`libcapsicum` and dependency `libsbuf`) impose a further 9% cost to the `fork()` syscall, presumably due to the additional virtual memory mappings being copied to the child process. This overhead is not present on `vfork()` which we plan to use in `libcapsicum` in the future; this suggests that a new `pdvfork()` would be a useful addition to the Capsicum system call repertoire, offering capability-oriented semantics for process creation with the reduced overhead of `vfork()`. Creating, synchronously sending an RPC to, and destroying a single sandbox (the “sandbox” label in Figure 5.12(b)) has a cost of about $1.5ms$, significantly higher than its subset components.

5.6.3 gzip performance

While the performance cost of `cap_enter()` is negligible compared to other activity, the cost of multi-process sandbox creation (already taken by `dhclient` and Chromium due to existing sandboxing) is significant.

Benchmark	Time/operation	Difference	% difference
`dup`	$1.061 \pm 0.000\mu s$	-	-
`cap_new`	$1.600 \pm 0.001\mu s$	$0.539 \pm 0.001\mu s$	$50.7\% \pm 0.08\%$
`shmfd`	$2.385 \pm 0.000\mu s$	-	-
`cap_new_shmfd`	$4.159 \pm 0.007\mu s$	$1.77 \pm 0.004\mu s$	$74.4\% \pm 0.181\%$
`fstat_shmfd`	$0.532 \pm 0.001\mu s$	-	-
`fstat_cap_shmfd`	$0.586 \pm 0.004\mu s$	$0.054 \pm 0.003\mu s$	$10.2\% \pm 0.506\%$
`read_1`	$0.640 \pm 0.000\mu s$	-	-
`cap_read_1`	$0.697 \pm 0.001\mu s$	$0.057 \pm 0.001\mu s$	$8.93\% \pm 0.143\%$
`read_10000`	$1.534 \pm 0.000\mu s$	-	-
`cap_read_10000`	$1.601 \pm 0.003\mu s$	$0.067 \pm 0.002\mu s$	$4.40\% \pm 0.139\%$
`write`	$0.576 \pm 0.000\mu s$	-	-
`cap_write`	$0.634 \pm 0.002\mu s$	$0.058 \pm 0.001\mu s$	$10.0\% \pm 0.241\%$
`cap_enter`	$1.220 \pm 0.000\mu s$	-	-
`getuid`	$0.353 \pm 0.001\mu s$	$-0.867 \pm 0.001\mu s$	$-71.0\% \pm 0.067\%$
`chroot`	$1.214 \pm 0.000\mu s$	$-0.006 \pm 0.000\mu s$	$-0.458\% \pm 0.023\%$
`setuid`	$1.390 \pm 0.001\mu s$	$0.170 \pm 0.001\mu s$	$14.0\% \pm 0.054\%$
`fork`	$268.934 \pm 0.319\mu s$	-	-
`vfork`	$44.548 \pm 0.067\mu s$	$-224.3 \pm 0.217\mu s$	$-83.4\% \pm 0.081\%$
`pdfork`	$259.359 \pm 0.118\mu s$	$-9.58 \pm 0.324\mu s$	$-3.56\% \pm 0.120\%$
`pingpong`	$309.387 \pm 1.588\mu s$	$40.5 \pm 1.08\mu s$	$15.0\% \pm 0.400\%$
`fork_exec`	$811.993 \pm 2.849\mu s$	-	-
`vfork_exec`	$585.830 \pm 1.635\mu s$	$-226.2 \pm 2.183\mu s$	$-27.9\% \pm 0.269\%$
`pdfork_exec`	$862.823 \pm 0.554\mu s$	$50.8 \pm 2.83\mu s$	$6.26\% \pm 0.348\%$
`sandbox`	$1509.258 \pm 3.016\mu s$	$697.3 \pm 2.78\mu s$	$85.9\% \pm 0.339\%$

Table 5.5: Micro-benchmark results for various system calls and functions, grouped by category.

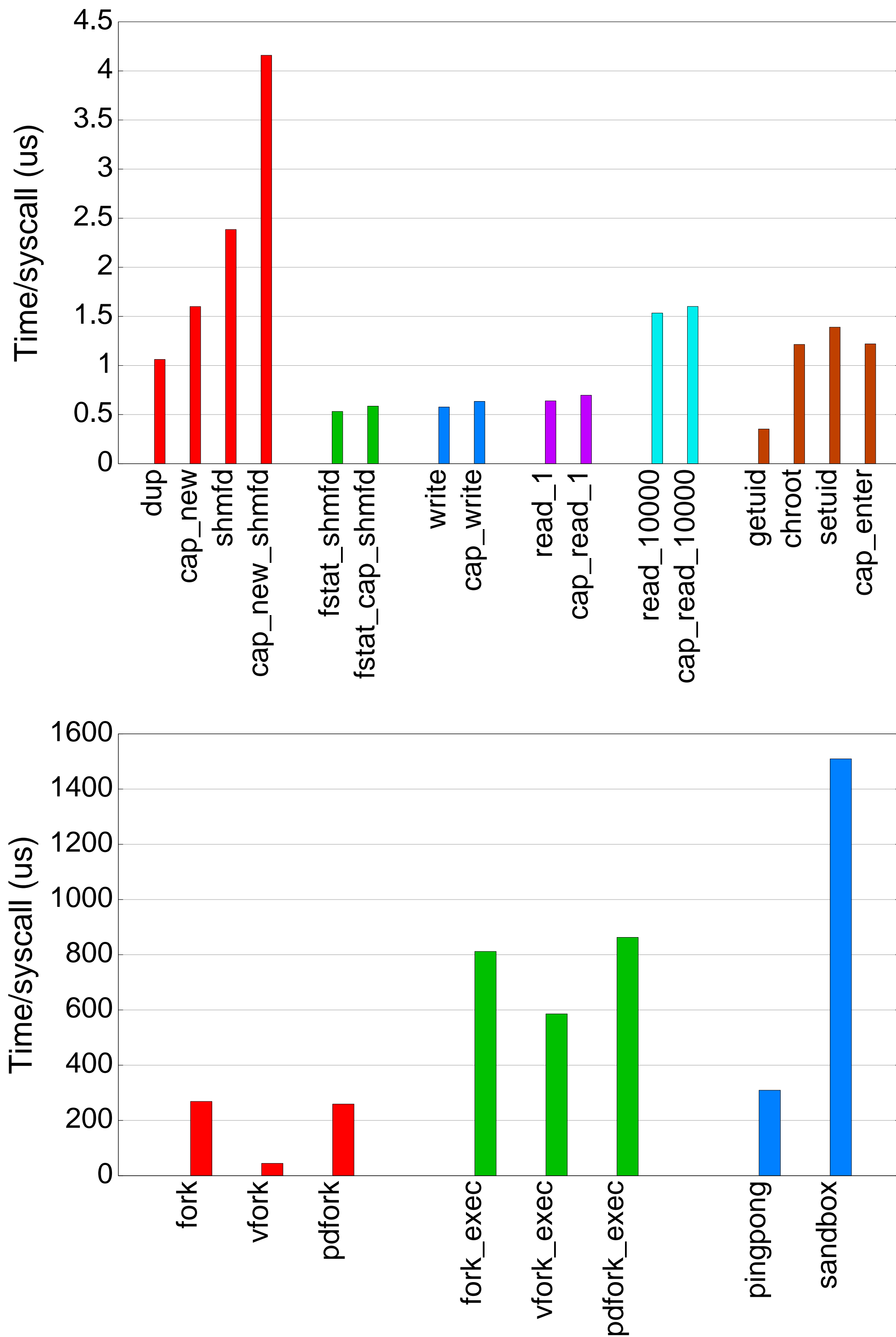

Figure 5.12: Capsicum system call performance compared to standard UNIX calls.

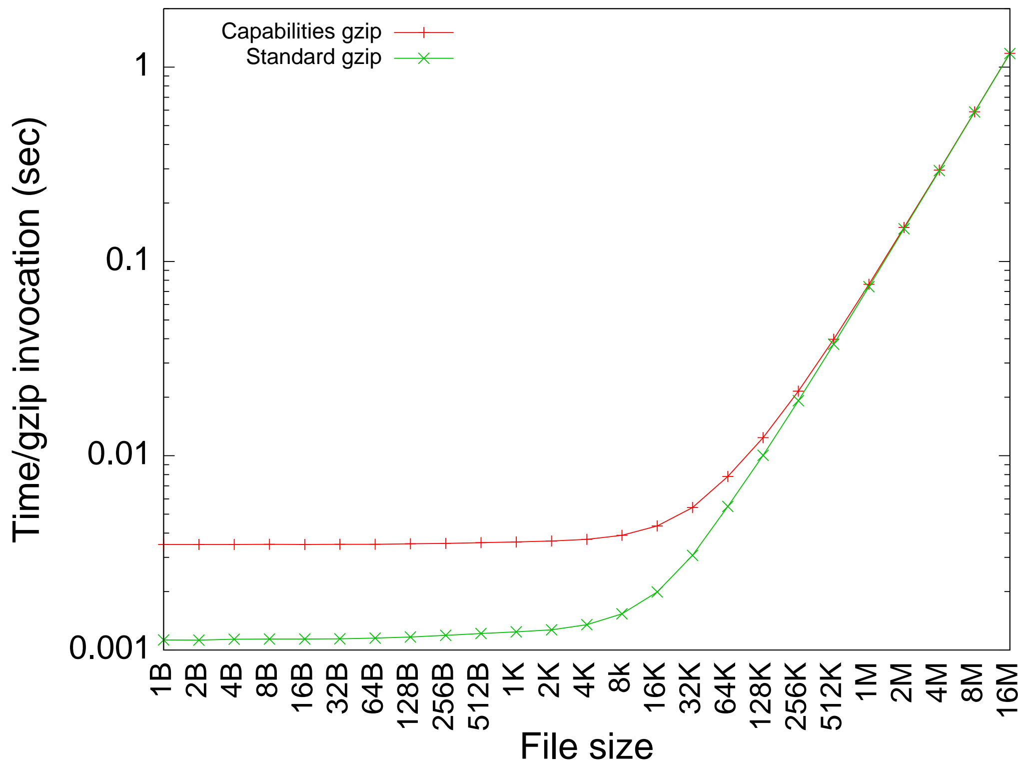

Figure 5.13: Run time per `gzip` invocation against random data, with varying file sizes; performance of the two versions come within 5% of one another at around a 512K.

To measure the cost of sandbox creation, we timed `gzip` compressing files of various sizes. Since the overheads of sandbox creation are purely at startup, we expect to see a constant-time overhead to the capability-enhanced version of `gzip`, with identical linear scaling of compression performance with input file size. Files were pre-generated on a memory disk by reading a constant-entropy data source: `/dev/zero` for perfectly compressible data, `/dev/random` for perfectly incompressible data, and base 64-encoded `/dev/random` for a moderate high entropy data source, with about 24% compression after gzipping. Using a data source with approximately constant entropy per bit minimises variation in overall `gzip` performance due to changes in compressor performance as files of different sizes are sampled. The list of files was piped to `xargs -n 1 gzip -c > /dev/null`, which sequentially invokes a new `gzip` compression process with a single file argument, and discards the compressed output. Sufficiently many input files were generated to provide at least 10 seconds of repeated `gzip` invocations, and the overall run-time measured. I/O overhead was minimised by staging files on a memory disk. The use of `xargs` to repeatedly invoke `gzip` provides a tight loop that minimising the time between `xargs`' successive `vfork()` and `exec()` calls of `gzip`. Each measurement was repeated 5 times and averaged.

Benchmarking `gzip` shows high initial overhead when compressing single-byte files, but also that the approach in which file descriptors are wrapped in capabilities and

delegated rather than using pure message passing leads to asymptotically identical behaviour as file size increases and constant startup costs are dominated by compression workload, which is unaffected by Capsicum. We find that the overhead of launching a sandboxed `gzip` is $2.37 \pm 0.01ms$, independent of the type of compression stream. This cost derives from several factors, including the added overhead of creating a new sandbox ($1.5ms$ in earlier micro-benchmarking), as well as additional costs in transferring file descriptors with an RPC, and linkage overheads due to the use of shared libraries in the sandbox. For many workloads, this one-off performance cost is negligible, or can be amortised by passing multiple files to the same `gzip` invocation.

5.7 Future work

Capsicum provides an effective platform for capability work on UNIX platforms. However, further research and development are required to bring this project to fruition.

We believe further refinement of the Capsicum primitives would be useful. Performance could be improved for sandbox creation, perhaps employing an Capsicum-centric version of the S-thread primitive proposed by Bittau. Further, a "logical application" operating system construct might improve termination properties.

Another area for research is in integrating user interfaces and OS security; Shapiro has proposed that capability-centered window systems are a natural extension to capability operating systems. Improving the mapping of application security constructs into OS sandboxes would also improve the security of Chromium, which currently does not consistently assign web security domains to sandboxes. It is in the context of windowing systems that we have found capability delegation most valuable: by driving delegation with UI behaviors, such as Powerboxes (file dialogues running with ambient authority) and drag-and-drop, Capsicum can support gesture-based access control research.

Finally, it is clear that the single largest problem with Capsicum and other privilege separation approaches is programmability: converting local development into de facto distributed development adds significant complexity to code authoring, debugging, and maintenance. Likewise, aligning security separation with application separation is a key challenge: how does the programmer identify and implement compartmentalisations that offer real security benefits, and determine that they've done so correctly? Further research in these areas is critical if systems such as Capsicum are to be used to mitigate security vulnerabilities through process-based compartmentalisation on a large scale.

5.8 Related work

As described in Chapter 1, capability system approaches to security were developed simultaneously with time-sharing systems such as MULTICS [113]. Capsicum adopts philosophical elements from a number of capability systems, but especially CMU's

Hydra [26] and Mach [3], making the capability model a first class component of the operating system's access control.

However, unlike pure capability systems, Capsicum both retains global namespaces (outside of capability mode), and differentiates capabilities for resources offered by the kernel and userspace. The former are Capsicum capabilities, providing native OS performance when directly accessing objects such as sockets and files – however, since they do not support interposition, certain desirable constructs, such as revocation, are not supported. The latter are based on IPC objects such as UNIX domain sockets, offering full message-passing semantics but the added cost of inter-process communication. Capsicum places significantly little emphasis on recrafting IPC primitives, such as those investigated by Shapiro in EROS [124] (now CapROS), inspired by KEYKOS [56].

Provos's OpenSSH privilege separation [103] and Kilpatrick's Privman [71] rekindled interest in microkernel-like compartmentalisation projects, such as the Chromium web browser [106] and Capsicum's logical applications. In fact, large application suites compare formidably with the size and complexity of monolithic kernels: the FreeBSD kernel is composed of 3.8 million lines of C, whereas Chromium and WebKit come to a total of 4.1 million lines of C++. How best to decompose monolithic applications remains an open research question; Bittau's Wedge offers a promising avenue of research in automated identification of software boundaries through dynamic analysis [20].

Seaborn and Hand have explored application compartmentalisation on UNIX through capability-centric Plash [119], and Xen [89], respectively. Plash offers an intriguing blend of UNIX semantics with capability security by providing POSIX APIs over capabilities, but is forced to rely on the same weak UNIX primitives analysed in Section 5.5. Supporting Plash on stronger Capsicum foundations would offer greater application compatibility to Capsicum users. Hand's approach suffers from similar issues to `seccomp`, in that the run-time environment for sandboxes is functionality-poor. Garfinkel's Ostia [49] also considers a delegation-centric approach, but focuses on providing sandboxing as an extension, rather than a core OS facility.

5.9 Conclusion

This chapter has described Capsicum, a practical capabilities extension to the POSIX API, and a prototype based on FreeBSD, planned for inclusion in FreeBSD 9.0; interest has also been expressed in integrating Capsicum into Google's ChromeOS and the NetBSD operating system. An evaluation of several real-world applications suggests that Capsicum is an effective tool in supporting application compartmentalisation and privilege separation, mitigating vulnerabilities by reducing the privilege delegated to sandboxes. The capability-centred model seems a more natural fit to models being adopted by programmers than existing sandbox techniques on UNIX systems.

Capsicum's relationship to existing access control and security techniques appears constructive: it usefully complements mandatory techniques in a manner not dissimilar

to the link between capabilities and mandatory access control found in systems such as DTMach [120] and LOCK [114]. There always exists a risk that the composition of security technologies leads to new security failures, which cannot be ignored – on the other hand, the relationship between mandatory access control and capabilities has been studied extensively in the research literature, giving us confidence that they can be used together successfully.

Capsicum also provides a useful tool for potential future work, such as compartmentalisation of further types of applications (such as office suites), exploring applications launched without any access ambient authority (rather than self-compartmentalising), delegation of capabilities through the user interface, and new techniques for program analysis and decomposition. Capsicum lends itself to adoption by blending immediate security improvements to current applications with the long-term prospects of a more capability-oriented future.

Chapter 6

Conclusions

It seems remarkable, almost fifty years after Corbató, Vyssotsky, Daley, and Neumann first described the Multics security architecture [28, 29], that the integration of access control into operating systems remains an open research problem. Although Saltzer and Schroeder's foundational principles of computer security showed remarkable insight [113], the convergence of technical, economic, and social circumstances supporting their widespread adoption has only recently occurred. Five broader trends in computing have combined to force renewed interest in those foundational principles: near-ubiquitous network connectivity; a clear and present threat of attack to even the most trivial use of computers; the increasing reliance of the world economy on electronic communication and transactions; orders of magnitude increases in software size and complexity; and dramatic changes in hardware performance. The central concern of this dissertation has been to demonstrate that "designing in" operating system security extensibility can help address both the immediate security needs of computer vendors, application writers, and end-users, while also providing a long-sought technology transfer path for decades of systems security research.

In Chapter 2, I developed not just new theory, but also practical exploit techniques for concurrency vulnerabilities previously considered only theoretical. These results motivated design choices throughout the remainder of my dissertation, and also caused several companies to change product designs to avoid the use of vulnerable techniques.

In Chapters 3 and 4, I described the multi-year process of designing, implementing, and deploying a new type of reference monitor – one not only resistant to concurrency attacks, but also able to support new flexibility for operating system access control, encouraging access control localisation and policy research. Through a series of case studies – the open source FreeBSD operating system, nCircle's enforcement appliance, and Apple's Mac OS X and iOS operating systems – I explored the impact of various design choices and continuing tensions between policy requirements and expressibility in widely-deployed open source and commercial products.

In Chapter 5, I investigated two concepts. First, I presented a new and practical blend of capability security (a neglected but powerful security design philosophy with

strong historic ties to Cambridge) and the UNIX process model, targeted at the burgeoning field of security-aware application writers. Second, and perhaps more critically, I developed the notion of a *hybrid capability architecture*, offering not just a vision, but also a practical transition path from current commodity systems to a markedly different architecture for operating system and application security.

Throughout, I have built on past research, developing new theory and new approaches that allow a rich field of access control research to be newly accessible to commodity system designs.

6.1 Principles

The following sections consider some of the principles and design philosophies that spanned the various research projects making up this dissertation.

6.1.1 Access control extensibility is a policy

While Saltzer and Schroeder's principles apply throughout the software stack, their application to operating systems is particularly important in order to ensure that higher software layers have secure foundations to build on. The diversity of security requirements and plethora of solutions proposed by the security research community has, however, proven a particular challenge to operating system developers. Each new model is promoted as a panacea, but proves instead to be a blend of difficult trade-offs: problems with usability, administrative complexity, application-programmer confusion, new vulnerabilities, and worst of all, significant performance overheads that, unlike security benefit, can be easily quantified. It has become increasingly clear that there can be no "one policy to rule them all": instead, we will live in a world in which many policies compete for deployment.

When I began my research into operating system security extensibility, I hoped that extensibility would address these concerns by allowing operating system vendors to avoid committing to any one solution, catering to the needs of a diverse audience through the flexibility of allowing sites and deployments to select their own specific balances between functionality, performance, and security. I also hoped that making it easier to experiment with new access control policies would encourage investigation leading to *better* access control models. Experience in researching, developing, and transferring the MAC Framework has born out both of these hopes.

Further, Capsicum suggests that access control extensibility is not only a way to avoid too narrow an access control focus, but is also a necessary tool for the construction of applications that come with their own security models. Increasingly applications, rather than operating systems, are the point of confluence for multiple security domains – from office suites handling documents from different origins to web browsers running code on behalf of different websites. System-centric access control models will remain central in protecting the trusted computing base (TCB) and enforcing global properties,

but facilities targeting application-local policy and enforcement are just as critical to ensuring robustness when mapping distributed policies into local enforcement.

6.1.2 Rehabilitating capabilities

The idea of a *capability* originates in Dennis and Van Horn's 1965 paper on multiprogramming [33], and up until the early 1990s, capability-centric system design philosophies were assumed to be the most desirable direction for high-assurance system designs. With the decline of capability hardware designs, and the rise and fall of microkernels, local capability models have seen only limited deployment (the notable exception is the Java Virtual Machine (JVM)).

Capsicum attempts to rehabilitate the capability design philosophy for mainstream operating systems by arguing that capabilities provide a better primitive for application-driven access control extension. Particularly, I believe that the delegation-centric philosophy of capabilities lends itself to non-traditional policy sources: application structure, user interface gestures, etc, by leaving the OS responsible for enforcement, but only some portions of policy. More generally, the delegation-centric enforcement model provided by capabilities seems a good match for an increasingly sandbox-focused security: applications are increasingly operating systems themselves, supporting (often malicious) mobile code embedded in a distributed security model. I see capabilities as complementing other access control paradigms, such as discretionary and mandatory access control, a view more consistent with historic capability system designs [96] than recent pure capability systems [124].

6.1.3 The risks of software interposition

My research into the concurrency implications of software interposition for security do not apply just to system call wrappers, as explored in Chapter 2: interposition is a software design construct used throughout security as it separates enforcement and policy from the implementation of an underlying service. I demonstrated that interposition in the presence of concurrency is not simply a question of sequentially calling hooks before an underlying software service – careful reasoning about the goals, and in some cases, synchronisation spanning layers of abstraction, is required for correctness. In non-security circumstances, naïve application of interposition leads to bugs; in the security context, I have illustrated how such bugs may easily be escalated to complete bypass of protections.

Philosophical concerns about interposition have far-reaching consequences. Proponents of object-capability systems argue for interposition as a method for implementing access control policies such as revocation [105, 62]. This has immediate consequences for systems such as Capsicum – while our focus has not been on interposition (and, in fact, trivial interposition is difficult for some of its kernel-backed objects), interposition for application-level capabilities is a natural design pattern to be deployed over it.

The MAC Framework itself can be framed as an interposition system, albeit one

carefully designed for correctness in the presence of concurrency. With the continuing explosion of highly concurrent systems, from local multiprocessing to distributed systems, further formal consideration of the properties of interposition is called for – and in the mean time, so is extreme caution.

6.1.4 Technology transfer is research

Operating system security research has a 50-year history, making it inevitable that past ideas will be rediscovered or reinvented; for example, access control policies and security mechanisms first proposed in the context of kernel/process separation modality are now seeing frequent repurposing in the world of virtual machine separation. Often, but not always, new proposals of old ideas revisit the assumptions of earlier work – particularly, the limitations that may have led them to be disregarded, and ideally demonstrate how a change in circumstances or approach allows those ideas to be applied to new problems. In effect, then, much interesting new operating system security research, due to its heavy grounding in past approaches, is actually research into technology transfer. With the fundamental changes in computer use identified throughout this dissertation, such reinvestigations are extremely valuable, and my own research has built on past research wherever possible.

In Capsicum, I have tried to take into account the many lessons of technology transfer from developing and deploying the MAC Framework: focus on an incremental adoption path, show sensitivity to performance and binary compatibility, do not under-emphasize the value of open source as a tech transfer methodology, and carefully consider real-world deployment scenarios. Investigating real-world deployment is time-consuming and detail-oriented, but as the failures of past operating systems show, the research gap is in how to create new technologies suited for technology transfer.

6.1.5 A hybrid design philosophy

The MAC Framework and Capsicum both implement what I refer to as *hybrid models*, in which new (or even old) security ideas are applied to existing systems in such a way that there is a clear incremental adoption path for the resulting design. The notion that it is possible to have the best of both worlds, both backwards compatibility and fundamental new security features, is a key tenet to my approach to technology transfer-centric research, and one I hope to explore further in the future. This is not to dismiss blue-sky systems research building from the ground up, which I consider fundamental in developing new approaches, but rather to argue that a hybrid design philosophy is an important way to take those ideas and apply them to existing systems in second generation research.

6.1.6 Open source infrastructure tech transfer

A key argument made in Chapter 4 is that open source has proven an extremely successful tech transfer strategy for the MAC Framework. Open source systems such

as FreeBSD and Linux increasingly provide the operating system foundations for embedded systems and appliances by providing a mature and affordable infrastructure grounded in non-competitive and non-differentiating technologies. Appliance builders do not compete on low-level OS design technologies – rather, they benefit from a common baseline and joint investment in infrastructure.

By improving the security of that baseline, a philosophy also espoused in the DARPA CHATS research programme, there is the opportunity to improve countless downstream products. Of course, open source systems, as with commercial systems, come with their own technology transfer challenges. For example, Chapter 4 provides a detailed exploration of the performance-related changes made to the MAC Framework as a result of community feedback.

6.2 Future work

This dissertation proposes solutions to a number of significant research problems in operating system security; however, it has inevitably led to further research questions, which I consider briefly in this section. Collectively, these suggest a sizeable operating system security research agenda for the future.

6.2.1 System call wrappers

The appeal of system call wrappers lies in large part in their simplicity: they impose controls on a well-defined interface independently of the implementation of the underlying system. As my research into concurrency vulnerabilities has shown, this appeal is deceptive – successful interposition involves not just sequential or nested invocations of software components, but instead requires a rich understanding of their semantic composition. Interposition cannot simply be dismissed, however: it is a fundamental primitive of software composition, so we require a better understanding of how to apply it correctly.

When encountering the problems I described in Chapter 2, many people suggest a transition to message passing – I counter that argument by observing that while this closes a class of races I refer to as *syntactic races*, it leaves open a larger and more subtle class of races I have described as *semantic races*: ones that arise not from race conditions on the values of arguments, but their interpretations. Such races invoke significant synchronisation problems, including concern about deadlock. Further research into the nature of these problems would be extremely valuable, as they potentially exist not just in with system call wrappers, but also in other interposition models in OS kernels and also application software.

6.2.2 The MAC Framework

In Chapters 3 and 4, I described the TrustedBSD MAC Framework, a DARPA research project I began before starting my PhD at Cambridge, and have continued work

on as part of my PhD research. The framework is now a mature reference monitor used in countless commercial products with great success – however, research continues throughout the technology transfer process, and there remain many further opportunities to improve the framework.

Most immediately, many downstream consumers have made improvements to the MAC Framework that should be analysed and selectively merged back into the authoritative copy in FreeBSD – for example, improvements found in the Mac OS X port. Some of these changes are minor, but others reflect fundamental philosophical changes of approach that should be investigated.

More generally, it is clear that the last word in access control policies has not yet been spoken, suggesting that the creation of new access control models remains a promising avenue for future research. Apple's focus on path-centric access control specification, similar to earlier thinking in DTE, deserves further consideration – especially in light of its potential for semantic conflicts not only with local UNIX file systems that do not consider paths a first class object, but also in distributed file systems that either incorporate UNIX semantics, have no defined semantics "on the wire," or have semantics significantly different from local semantics. A clear moral here, and one seen also in Capsicum, is that security policy must be expressed in the vocabulary of the end user, referring to objects and actions that fit the user's mental model for system operation. Capsicum suggests other important sources of policy: application structure and user interaction – further investigation of the composition of mandatory access control models and capability models is likewise desirable.

I remain unsatisfied with the integration of security event auditing with the MAC Framework; while the framework can usefully control audit operation, how best to allow policies to generate audit records, annotating existing events with extended access control information, is an unsolved problem. I have speculated in FreeBSD design discussions that allowing policies to generate their own records, cross-referenced with base kernel records, may be one path forward.

6.2.3 Capsicum

Capsicum, significantly less mature in a technology transfer sense than the MAC Framework, raises many more research questions.

While it appears to be a significant improvement on the state-of-the-art, Capsicum relies on a number of worrying premises, not least that application writers can usefully perform compartmentalisation on complex applications using today's programming languages and runtime models. Compartmentalised applications, when authored in the C language and executing in the UNIX process model, are fundamentally distributed applications: the programmer is reduced to using message passing, which is not only markedly slower than direct function invocation, but also introduces significant programming hurdles due to the loss of an assumption of a single address space.

Research into how to perform compartmentalisation and improved models for sep-

aration are critical to the future success of Capsicum, but also more generally the application of the notion of the principle of least privilege. Can we create tools to help application authors identify decompositions of applications in support of specific and well-defined security policies? Can we provide tools to partially or even fully automate the compartmentalisation process? Can we develop new models for compartmentalisation, grounded in CPU and OS security features, that improve programmability for compartmentalised applications, refining the UNIX process model? How should programmers trade off security, performance, and programmability, and can the selection of trade-offs be made dynamic, allowing reworking of application compartmentalisation on demand as new vulnerabilities are discovered, or as the threat of attack goes up? Collectively, these research questions are critical to future work in operating system and application security.

6.2.4 CRASH-worthy Trustworthy Systems R&D (CTSRD)

Peter Neumann and I hope to pursue many of these research questions in a new DARPA-sponsored joint research project between SRI and Cambridge, CRASH-worthy Trustworthy Systems R&D (CTSRD). Among the research themes of the project is the development of Capability Hardware Enhanced RISC Instructions (CHERI), which translates the hybrid operating system design philosophy of Capsicum into a hybrid hardware design, blending commodity hardware designs based on page-oriented virtual memory with a delegation-oriented hardware capability model. This will allow efficient compartmentalisation of software, from application programs to operating system kernels (including Capsicum), by allowing security differentiation within a single address space. We hope that the results will solve core performance and programmability problems that currently hamper the use of compartmentalisation.

Another research them in CTSRD is a new invariants and testing system, Temporally Enhanced Security Logic Assertions (TESLA), which will help to address a key observation from the MAC Framework project: security policy enforcement, as embodied in a software implementation, is an artefact of the policy, rather than the policy itself. Similarly, existing enforcement and testing systems rely on instantaneous state of a system, but security properties are fundamentally temporal in notion. TESLA enhances existing assertions in the FreeBSD kernel and userspace to allow the expression of temporal rules, both in a temporal assertion logic inline with source code, and via explicit automata describing permissible state transitions. Initially, our goal will be to make testing security properties such as "check before use" and "eventual audit" easier; later, we hope to investigate using TESLA to drive mechanical implementation of enforcement, avoiding problems such as potential missing MAC Framework entry points, and creating a closer link between the policy and its implementation.

Bibliography

[1] 318 Inc. A brief introduction to Mac OS X SandBox Technology. `http://techjournal.318.com/security/a-brief-introduction-to-mac-os-x-sandbox-technology/`, April 2008. 30, 121

[2] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th NIST-NCSC National Computer Security Conference*, pages 135–143, 1990. 29, 89

[3] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, August 1986. 24, 43, 160

[4] W. B. Ackerman and W. W. Plummer. An implementation of a multiprocessing computer system. In *SOSP '67: Proceedings of the first ACM Symposium on Operating System Principles*, pages 5.1–5.10, New York, NY, USA, 1967. ACM. 23

[5] J. P. Anderson. Computer Security Technology Planning Study. Technical report, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01730, October 1972. 16, 17, 22, 34, 89

[6] R. Anderson and S. Fuloria. Certification and evaluation: a security economics perspective. In *ETFA'09: Proceedings of the 14th IEEE international conference on emerging technologies & factory automation*, pages 1156–1162, Piscataway, NJ, USA, 2009. IEEE Press. 23

[7] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. 20

[8] G. R. Andrews. Partitions and principles for secure operating systems. Technical report, Cornell University, Ithaca, NY, USA, 1975. 25

[9] Apple Inc. Kernel Authorization. Technical Note TN2127, Apple Computer, Inc., 2007. `http://developer.apple.com/technotes/tn2005/tn2127.html`. 30, 44, 89, 112, 125

[10] Apple Inc. iOS Dev Center. `http://developer.apple.com/devcenter/ios/index.action`, 2010. 51, 106

[11] Apple Inc. Mac OS X Snow Leopard. `http://www.apple.com/macosx/`, 2010. 51, 106

[12] Argus Systems Group. Pitbull foundation. `http://www.argussystems.com/Products/pitbull-foundation.html`, October 2010. 20, 52

[13] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical Domain and Type Enforcement for UNIX. In *SP '95: Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 66, Washington, DC, USA, 1995. IEEE Computer Society. 21, 52, 89, 116

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM. 21

[15] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973. 14, 20, 47, 89

[16] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Trans. Softw. Eng.*, 16(6):608–618, 1990. 21, 109

[17] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM. 27

[18] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, Apr. 1977. 14, 20, 47, 75, 80, 89

[19] R. Bisbey and D. Hollingworth. Protection Analysis: Final Report. Technical Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978. 18

[20] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2008. 160

[21] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of 7th DOD/NBS Computer Security Conference*, pages 291–293, September 1984. 20

[22] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *In Proceedings of the 8th National Computer Security Conference*, 1985. 21, 47, 89

[23] M. Branstad and J. Landauer. Assurance for the Trusted Mach operating system. *Proceedings of the Fourth Annual Conference on Computer Assurance*, pages 103–108, 1989. 26

[24] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004. USENIX Association. 27, 60

[25] R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the great firewall of china. In *PETS '06: Proceedings of the 8th international symposium on Privacy Enhancing Technologies*, Berlin, Heidelberg, 2006. Springer-Verlag. 11

[26] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 141–160, New York, NY, USA, 1975. ACM. 17, 23, 57, 160

[27] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *AIEE-IRE '62 (Spring): Proceedings of the May 1–3, 1962, Spring Joint Computer Conference*, pages 335–344, New York, NY, USA, 1962. ACM. 17

[28] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, New York, NY, USA, 1965. ACM. 17, 163

[29] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 213–229, New York, NY, USA, 1965. ACM. 20, 163

[30] P. J. Dawidek and S. Zak. CerbNG: system firewall mechanism, 2007. `http://cerber.sourceforge.net/`. 33

[31] D. Dean and A. J. Hu. Fixing Races for Fun and Profit: How to use `access(2)`. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, August 2004. 35

[32] P. J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976. 24

[33] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966. 17, 23, 165

[34] Different Internet Experience Ltd. Khobe 8.0 earthquake for windows desktop security software. `http://www.matousec.com/info/articles/khobe-8.0-earthquake-for-windows-desktop-security-software.php`. 44

[35] E. Efrat. kauth: kernel authorization framework. `http://www.netbsd.org/~elad/recent/man/kauth.9.html`, January 2007. 44, 89

[36] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39:17–30, October 2005. 26, 51

[37] J. Epstein, J. McHugh, and R. Pascale. Evolution of a Trusted B3 Window System Prototype. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, USA, May 1992. 21

[38] R. S. Fabry. The case for capability based computers (extended abstract). In *SOSP '73: Proceedings of the fourth ACM Symposium on Operating System Principles*, page 120, New York, NY, USA, 1973. ACM. 25

[39] R. Farrow. A Report on the Linux 2.5 Kernel Developers Summit. *;login:*, 26(3), June 2001. 29

[40] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. 17

[41] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. 21

[42] C. Fetzer and M. Süßkraut. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 273–286, New York, NY, USA, 2008. ACM. 44

[43] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association. 44

[44] Ford Aerospace and Communications Corporation. Secure Minicomputer Operating System (KSOS) Executive Summary: Phase I: Design of the Department of Defense Kernelized Secure Operating System. Technical report, 3939 Fabian Way, Palo Alto, CA 94303, March 1978. 25

[45] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. IEEE, 2000. 20, 35, 50, 53

[46] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999. 33, 53, 89

[47] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000. 88

[48] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003. 33

[49] T. Garfinkel, B. Pfa, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Internet Society*, 2003. 43, 160

[50] D. P. Ghormley, D. Patrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, June 1998. 33

[51] L. Gong and G. Ellison. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Pearson Education, 2003. 21, 26

[52] Google, Inc. Security and Permissions - Android Developers Dev Guide. `http://developer.android.com/guide/topics/security/security.html`, October 2010. 20

[53] Google, Inc. The Chromium Project: Design Documents: OS X Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`, October 2010. 152

[54] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. 26

[55] R. M. Graham. Protection in an information processing utility. In *SOSP '67: Proceedings of the first ACM Symposium on Operating System Principles*, New York, NY, USA, 1967. ACM. 17

[56] N. Hardy. Keykos architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985. 26, 160

[57] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Trans. Comput. Syst.*, 12(1):58–89, 1994. 53

[58] Information Systems Security Organization, National Security Agency. Controlled access protection profile version 1.d (CAPP), October 1999. 23

[59] Information Systems Security Organization, National Security Agency. Labeled security protection profile version 1.b (LSPP), October 1999. 23

[60] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. 27

[61] K. Johnson and M. Deksters. sysjail: systrace userland virtualization. `http://sysjail.bsd.lv/`, 2007. 39, 41

[62] A. K. Jones and W. A. Wulf. Towards the design of secure systems. *Software Practice and Experience*, 5(4):321–336, 1975. 24, 28, 165

[63] R. Jones. The temporary netperf homepage. `http://www.netperf.org/`, October 2010. 86

[64] R. Y. Kain and C. E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2):202–207, 1987. 20

[65] P. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000. 41, 102

[66] P. Kamp and R. N. M. Watson. Building systems to be shared, securely. *ACM Queue*, 2(5):42–51, 2004. 11

[67] P. A. Karger. Limiting the damage potential of discretionary trojan horses. In *IEEE Symposium on Security and Privacy*, pages 32–37, 1987. 29

[68] P. A. Karger. Using registers to optimize cross-domain call performance. *SIGARCH Computer Architecture News*, 17(2):194–204, 1989. 19

[69] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical report, HQ Electronic Systems Division: Hanscom AFB, MA., 1974. 17

[70] J. Kelly, W. Araujo, and K. Banerjee. Rapid service creation using the JUNOS SDK. *SIGCOMM Comput. Commun. Rev.*, 40(1):56–60, 2010. 51, 100

[71] D. Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference*, pages 273–284. USENIX Association, 2003. 160

[72] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and Countering System Intrusions Using Software Wrappers. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, August 2000. 39

[73] B. W. Lampson. Dynamic protection structures. In *AFIPS '69 (Fall): Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, pages 27–38, New York, NY, USA, 1969. ACM. 17

[74] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974. 17, 25

[75] B. W. Lampson. Redundancy and Robustness in Memory Protection. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Hardware II, pages 128–132. North-Holland, Amsterdam, 1974. 17

[76] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 132–140, New York, NY, USA, 1975. ACM. 23, 28

[77] S. B. Lipner, W. A. Wulf, R. R. Schell, G. J. Popek, P. G. Neumann, C. Weissman, and T. A. Linden. Security kernels. In *AFIPS '74: Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, pages 973–980, New York, NY, USA, 1974. ACM. 25

[78] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 29–42. USENIX Association, June 2001. 21, 89, 98, 152

[79] McAfee Inc. McAfee Firewall Enterprise. `http://www.mcafee.com/us/enterprise/products/network_security/firewall_enterprise.html`, October 2010. 21, 51, 121

[80] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference*, Berkeley, CA, USA, 1993. USENIX Association. 27

[81] M. K. McKusick. Enhancements to the fast filesystem to support multi-terabyte storage systems. In *BSDC'03: Proceedings of the BSD Conference 2003 on BSD Conference*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association. 51

[82] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004. 27, 92

[83] A. Mettler and D. Wagner. Class properties for security review in an object-capability subset of Java. In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–7, New York, NY, USA, 2010. ACM. 26

[84] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. 24, 26

[85] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`. 26

[86] T. C. Miller. Sudo, 2007. `http://www.gratisoft.us/sudo/`. 39, 40

[87] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973. 23

[88] S. J. Murdoch and R. N. M. Watson. Metrics for Security and Performance in Low-Latency Anonymity Systems. In *PETS '08: Proceedings of the 8th international symposium on Privacy Enhancing Technologies*, pages 115–132, Berlin, Heidelberg, 2008. Springer-Verlag. 11

[89] D. G. Murray and S. Hand. Privilege Separation Made Easy. In *Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC)*, pages 40–46. ACM, 2008. 160

[90] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997. 26

[91] National Computer Security Center (NCSC). *Trusted Computer System Evaluation Criteria (TCSEC)*. U. S. Department of Defense, December 1985. 22

[92] National Security Agency. NetTop. `http://www.nsa.gov/research/tech_transfer/fact_sheets/nettop.shtml`, October 2010. 22

[93] nCircle Network Security. `http://www.ncircle.com/`, October 2010. 51, 101, 121

[94] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI '96: Proceedings of the second USENIX symposium on Operating Systems Design and Implementation*, pages 229–243, New York, NY, USA, 1996. ACM. 27

[95] P. G. Neumann. Principled assuredly trustworthy composable architectures. Technical report, Computer Science Laboratory, SRI International, Menlo Park, December 2004. 48

[96] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A Provably Secure Operating System: The System, Its Applications, and Proofs, Second Edition. Technical Report CSL-116, Computer Science Laboratory, SRI International, May 1980. 17, 25, 89, 132, 165

[97] NIST Common Criteria Implementation Board. Common criteria version 2.1 (ISO IS 15408), 2000. 22

[98] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, New York, NY, USA, 2008. ACM. 44

[99] A. Ott. Rule Set Based Access Control (RSBAC) for Linux. `http://www.rsbac.org/`, October 2010. 29, 89

[100] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press. 24

[101] S. Potter, J. Nieh, and M. Selsky. Secure isolation of untrusted legacy applications. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association. 44

[102] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003. USENIX Association. 33

[103] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003. 133, 160

[104] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *SOSP '81: Proceedings of the eighth ACM Symposium on Operating Systems Principles*, pages 64–75, New York, NY, USA, 1981. ACM. 24

[105] D. Redell and R. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 197–209, August 1974. 29, 165

[106] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232, New York, NY, USA, 2009. ACM. 150, 160

[107] D. Rémy and J. Vouillon. Objective ML: a simple object-oriented extension of ML. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, New York, NY, USA, 1997. ACM. 27

[108] L. Riek and R. Watson. The age of avatar realism. *Robotics and Automation Magazine, IEEE*, 17(4):37 –42, Dec 2010. 11

[109] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974. 17

[110] Ruby Users Group. Ruby Programming Language. `http://www.ruby-lang.org/`, October 2010. 27

[111] J. M. Rushby. Design and verification of secure systems. In *SOSP '81: Proceedings of the eighth ACM Symposium on Operating Systems Principles*, pages 12–21, New York, NY, USA, 1981. ACM. 17, 25

[112] J. H. Saltzer. Protection and control of information sharing in Multics. In *SOSP '73: Proceedings of the fourth ACM Symposium on Operating System Principles*, New York, NY, USA, 1973. ACM. 17, 18

[113] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. volume 63, pages 1278–1308, September 1975. 17, 159, 163

[114] O. Sami Saydjari. Lock: an historical perspective. In *Proceedings of the 18th Annual Computer Security Applications Conference*. IEEE Computer Society, 2002. 21, 161

[115] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*. USENIX Association, June 1985. 48

[116] W. L. Schiller. The design and specification of a security kernel for the PDP-11/45. Technical report, The MITRE Corporation, Bedform, MA, 01730, March 1975. 25

[117] M. D. Schroeder. Engineering a security kernel for Multics. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 25–32, New York, NY, USA, 1975. ACM. 25

[118] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972. 19

[119] M. Seaborn. Plash: tools for practical least privilege, 2007. http://plash.beasts.org/. 43, 149, 160

[120] E. J. Sebes. Overview of the architecture of Distributed Trusted Mach. In *Proceedings of the USENIX Mach Symposium*, pages 20–22. USENIX Association, November 1991. 26, 108, 161

[121] Seccuris Inc. Assured Protection. http://www.seccuris.com/, October 2010. 51, 101

[122] SecurityFocus. Linux capabilities vulnerability. http://www.securityfocus.com/bid/1322, June 2000. 56, 74, 102

[123] SGI. Multilevel Security (MLS) by Trusted IRIX. http://www.sgi.com/pdfs/3241.pdf, 2002. 52

[124] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, Dec 1999. 26, 132, 160, 165

[125] D. Smørgrav. OpenPAM. http://www.openpam.org/, October 2010. 51

[126] C. Smowton. Secure 3D graphics for virtual machines. In *EUROSEC '09: Proceedings of the Second European Workshop on System Security*, pages 36–43, New York, NY, USA, 2009. ACM. 21

[127] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX Association. 26, 29, 44, 51, 108

[128] B. Spengler. Linux 2.6.30+/SELinux/RHEL5 test kernel 0day, exploiting the unexploitable. http://lwn.net/Articles/341773/, July 2009. 19

[129] Sun Microsystems. Trusted Solaris 8 Operating Environment: A Technical Overview, 2000. 21, 52

[130] C. Vance, T. C. Miller, R. Dekelbaum, and A. Reisse. Security-Enhanced Darwin: Porting SELinux to Mac OS X. In *Proceedings from the Third Annual Security Enhanced Linux Symposium*, 2007. Draft 2007/01/22 11:35. 51, 108

[131] C. Vance and R. N. M. Watson. Security Enhanced BSD. Technical report, Network Associates Laboratories, 2003. 21, 44, 98

[132] T. V. Vleck. The IBM 360/67 and CP/CMS. http://www.multicians.org/thvv/360-67.html. 21

[133] I. VMWare. VMWare Virtualization Software. http://www.vmware.com/, October 2010. 21

[134] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM. 19, 27

[135] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980. 25

[136] S. T. Walker. The advent of trusted computer operating systems. In *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference*, pages 655–665, New York, NY, USA, 1980. ACM. 25

[137] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association. 28

[138] R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, June 2001. 107

[139] R. N. M. Watson. Introduction to multithreading and multiprocessing in the FreeBSD SMPng network stack. In *Proceedings of EuroBSDCon 2005*, November 2005. 11

[140] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT '07: Proceedings of the first USENIX Workshop on Offensive Technologies*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association. 11

[141] R. N. M. Watson. How the FreeBSD project works. In *Proceedings of EuroBSDCon 2008*, October 2008. 11

[142] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, Berkeley, CA, USA, 2010. USENIX Association. 11

[143] R. N. M. Watson, B. Feldman, A. Migus, and C. Vance. Design and Implementation of the TrustedBSD MAC Framework. In *Proceedings of the Third DARPA Information Survivability Conference and Exhibition (DISCEX), IEEE*, April 2003. 11, 29, 33, 44, 48, 74, 152

[144] R. N. M. Watson and W. Salamon. The FreeBSD Audit System. In *Proceedings UKUUG LISA Conference*. UKUUG, 2006. 11, 23, 124

[145] R. N. M. Watson and C. Vance. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *In USENIX Annual Technical Conference*, pages 285–296. USENIX Association, 2003. 11

[146] M. V. Wilkes and R. M. Needham. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1979. 17, 19, 23

[147] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for Linux using Mondriaan memory protection. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 31–44, New York, NY, USA, 2005. ACM. 19

[148] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002. 29, 44, 51, 89

[149] Q. Wu, H. Dai, X. Liu, and H. Feng. The Kylin operating system. In *Proceedings of EuroBSDCon 2006*. WillyStudios.com, November 2006. 101

[150] X. Wu, Z. Zhou, Y. He, and H. Liang. Static Analysis of a Class of Memory Leaks in TrustedBSD MAC Framework. *Information Security Practice and Experience*, pages 83–92, 2009. 99, 121

[151] X. Wu, Z. Zhou, Y. He, H. Liang, and C. Yuan. Static analysis based correctness verification for mandatory access control framework. *Jisuanji Xuebao(Chinese Journal of Computers)*, 32(4):730–739, 2009. 99, 121

[152] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974. 17

[153] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society. 27

[154] K. Yee, M. Miller, and J. Shapiro. Capability myths demolished. `http://zesty.ca/capmyths/`, October 2010. 21

[155] E. Zadok and J. Nieh. FiST: a language for stackable file systems. In *ATEC '00: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association. 53

[156] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association. 26