DOI:10.1145/2408776.2408792

Article development led by acmqueue
queue.acm.org

Open source security foundations for mobile and embedded devices.

BY ROBERT N.M. WATSON

A Decade of OS Access-Control Extensibility

TO DISCUSS OPERATING-SYSTEM security is to marvel at the diversity of deployed access-control models: Unix and Windows NT multiuser security, Type Enforcement in SELinux, anti-malware products, app sandboxing in Apple OS X, Apple iOS, and Google Android, and application-facing systems such as Capsicum in FreeBSD. This diversity is the result of a stunning transition from the narrow 1990s Unix and NT status quo to *security localization*—the adaptation of operating-system security models to site-local or product-specific requirements.

This transition was motivated by three changes: the advent of ubiquitous Internet connectivity; a migration from dedicated embedded operating systems to general-purpose ones in search of more sophisticated software stacks; and widespread movement from multiuser computing toward single-user devices with complex application models. The transition was facilitated by *extensible access-control frameworks*, which allow operating-system kernels to be more easily adapted to new security requirements.

One such extensible kernel reference-monitor framework is the TrustedBSD MAC (Mandatory Access Control) Framework, developed beginning in 2000 and shipped in the open source FreeBSD operating system in 2003. This article first describes the context and challenges for access-control extensibility and high-level framework design, then turns to practical experience deploying security policies in several framework-based products, including FreeBSD, nCircle appliances, Juniper's Junos, and Apple's OS X and iOS. While extensibility was key to each of these projects, they motivated considerable changes to the framework itself, so the article also explores how the framework did (and did not) meet each product's requirements, and finally reflects on the continuing evolution of operating-system security.

A Quiet Revolution in OS Design

Embedded and mobile operating systems have changed greatly in the past 20 years: devices have gained the CPU power to run general-purpose operating systems; they have been placed in ubiquitous networking environments; they have needed to support mature software stacks including third-party applications; and they have found themselves exposed to malicious activity motivated by strong financial incentives. Vendors built on existing operating systems—often open source—to avoid creating them from scratch. This provided mature application frameworks and complex network stacks, both areas of weakness for then-contemporary "embedded operating systems." One early example is Juniper's Junos, a version of FreeBSD adapted for router control planes in 1998. This trend had come to fruition by 2007 when Google's Android, based

on Linux, and Apple's iOS, based in part on Mach and FreeBSD, became available, transforming the smartphone market.

Common to all of these environments is a focus on security and reliability: as third-party applications are deployed in systems from Junos, via its SDK, and to iOS/Android app stores, sandboxing becomes critical, first to prevent *bricking* (reducing a device to a mere brick as a result of malfunction or abuse) and later to constrain malware. This trend is reinforced by mobile-phone access to online purchasing, and most recently, banking and payment systems. As a result, the role of operating-system security has shifted from protecting multiple users from each other toward protecting a single operator or user from untrustworthy applications. In 2013, embedded devices, mobile phones, and tablets are points of confluence: the interests of many different parties—consumers, phone vendors, application authors, and online services—must be mediated with the help of operating systems that were designed for another place and time.

Access-Control Frameworks. Operating-system developers must satisfy device vendors, who require everything from router and firewall hardening to mobile-phone app sandboxing. Operating-system vendors had accurately observed a difficult adoption path for historic *trusted operating systems*, whose mandatory access-control schemes suffered from poor usability, performance, maintainability, and—perhaps most critically—end-user demand. Likewise, they saw many promising new security models in research, each with unknown viability, suggesting that no single access-control model would meet all needs. This practical reality of security localization directly motivates extensible access control.

Research over the preceding 20 years had made clear the need for a *reference monitor*—a self-contained, nonbypassable, and compact (hence verifiable) centralization of access control.[2] By the early 1990s, this concept had been combined with the notion of *encapsulation*, appearing in Abrams et al.'s Generalized Framework for Access Control (GFAC),[1] and by the late 1990s in Ott's Rule Set-based Access Control (RSBAC)[14] and Spencer et al.'s Flask security architecture.[17] Mainstream operating-system vendors did not adopt these approaches until the early 2000s with the MAC Framework on FreeBSD[22] and shortly after, Linux Security Modules (LSM).[23] In both cases, a key concern was supporting third-party security models without committing to fixed policies as had earlier trusted systems.

The MAC Framework

The MAC Framework was proposed in 1999, with the first whitepaper on its design published in June 2000.[20] It appeared in FreeBSD 5.0 in 2003 as an experimental feature—compiled out by default but available to early adopters. FreeBSD 8.0 in 2009 included the framework as a production feature, compiled into the default kernel. (A timeline of key events in its development appears in Figure 1.)

The MAC Framework offers a logical solution to the problem of kernel access-control augmentation: extension infrastructure able to represent many different policies, offering improved maintainability and supported by the operating-system vendor. Similar to device drivers and virtual file system (VFS) modules,[10] policies are compiled into the kernel or loadable modules and implement well-defined kernel programming interfaces (KPIs). Policies can augment access-control decisions and make use of common infrastructure such as object labeling to avoid direct kernel modification and code duplication. They are able to enforce access control across a broad range of object types, from files to network interfaces, and integrate with the kernel's concurrency model.

Mandatory Policies. MAC describes a class of security models in which policies constrain the interactions of all system users. Whereas discretionary access control (DAC) schemes such as file-system access-control lists

Figure 1. MAC Framework research and development with key corporate contributions.

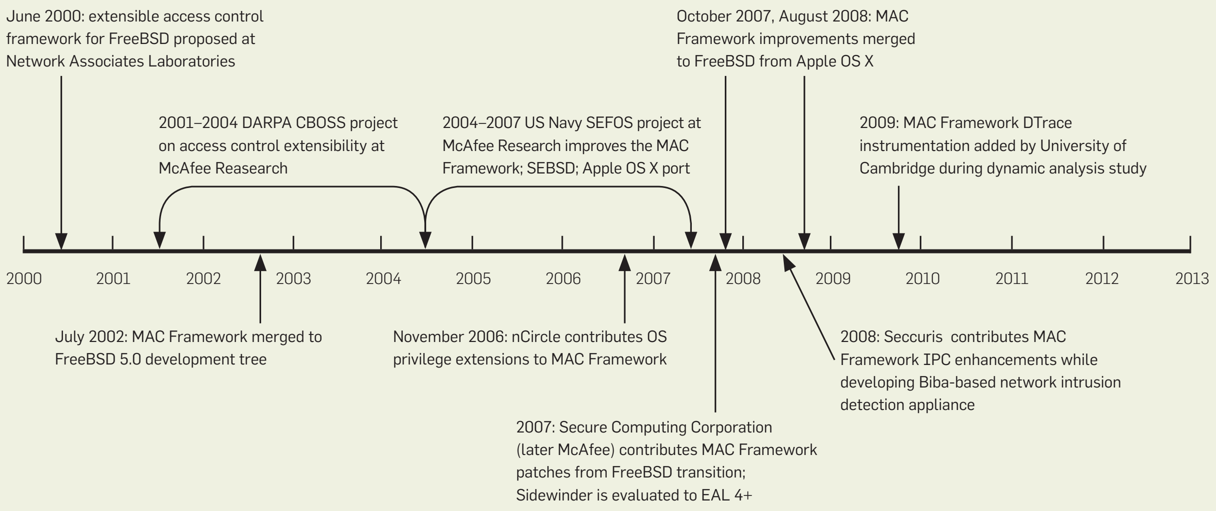
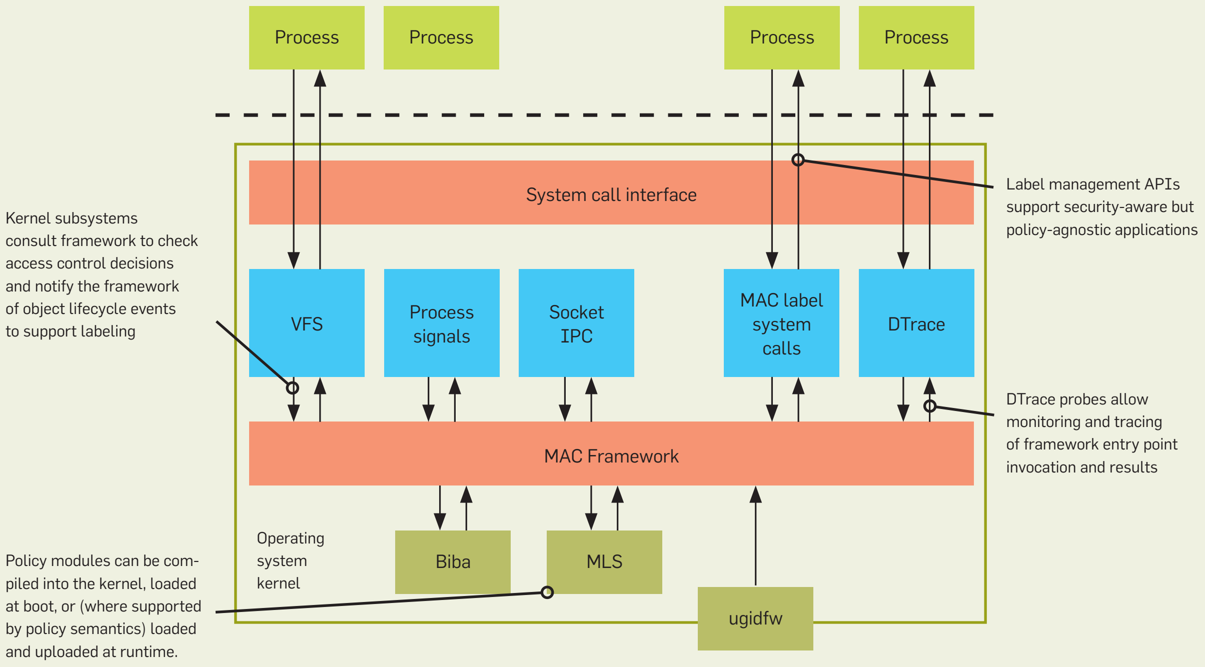


Figure 2. Policy models are encapsulated in kernel modules that augment kernel access control.


Process
Process
Process
Process
System call interface
Label management APIs support security-aware but policy-agnostic applications
Kernel subsystems consult framework to check access control decisions and notify the framework of object lifecycle events to support labeling
VFS
Process signals
Socket IPC
MAC label system calls
DTrace
DTrace probes allow monitoring and tracing of framework entry point invocation and results
MAC Framework
Operating system kernel
Biba
MLS
ugidfw
Policy modules can be compiled into the kernel, loaded at boot, or (where supported by policy semantics) loaded and uploaded at runtime.

(ACLs) allow object owners to protect (or share) objects at their own discretion, MAC enforces systemwide security invariants regardless of user preference. The research literature describes a plethora of mandatory policies grounded in information flow and rule-based models.

Early mandatory policies focused on information flow, requiring ubiquitous enforcement throughout the kernel. *Multilevel security* (MLS) protects confidentiality by labeling user clearance and data confidentiality, limiting flow.[5] The *Biba integrity policy* is the logical dual of MLS, protecting integrity.[6] These models maintain subject and object *security labels* holding confidentiality or integrity information, and controlling operations that might lead to information upgrade or downgrade.

SRI International's PSOS (Provably Secure Operating System) design included strong enforcement of object types, supplementing capability protections.[13] This evolved into Boebert's Type Enforcement (TE)[7] and Badger et al.'s Domain and Type Enforcement (DTE),[4] which have proven influential, with TE deployed in SELinux[11] and McAfee's Sidewinder firewall. Both models are flexible and fine-grained, labeling subjects and objects with symbolic domains and types. Administrator-controlled rules authorize interactions and transitions between domains.

Finally, a broad class of product-specific *hardening policies* is also relevant; these take less principled approaches, offering direct control over services rather than abstract models.

Before Access-Control Extensibility. In implementation papers, we critiqued contemporaneous techniques from experience:

- *Direct kernel modification* was used for most trusted systems, whether originated by operating-system vendors (for example, Trusted Solaris) or third-party extensions (for example, Argus Pitbull). Tracking upstream operating-system development is problematic: extensions are unable to depend on public, and hence more stable, APIs (application programming interfaces) and KPIs—and less obvious at the time, ABIs (application binary interfaces) and KBIs (kernel binary interfaces). Upstream churn frequently triggers design and source-code conflicts with security extensions. Assurance is also affected, as the burden of arguing for correctness is left entirely in the hands of the extension writer.

- *System call interposition* is widely used in antivirus systems and, in the past, security extension products and research systems.[9] Kernel concurrency proves a particular challenge, and we have demonstrated easily exploited race conditions between wrappers and kernels.[19]

The MAC Framework offers a logical solution to the problem of kernel access-control augmentation: extension infrastructure able to represent many different policies, offering improved maintainability and supported by the operating-system vendor.

Guiding Design Principles. The dual goals of access-control extensibility and encouraging upstream and downstream vendor engagement motivated several design principles for the MAC Framework:

Do not commit to a specific access-control policy. There is no consensus on a single policy or even policy language; instead, capture policy models in C code.

Avoid policy-specific intrusions into the kernel. Encapsulate internals behind policy-agnostic interfaces. This leads naturally to object-centered design—access-control checks with respect to subjects, objects, and methods.

Provide policy-agnostic infrastructure. This satisfies common requirements beyond access-control instrumentation, such as labeling and tracing.

Support multiple simultaneously loaded policies. In this way different aspects of policy, perhaps from different vendors, can be independently expressed. For example, Trusted IRIX and Argus Pitbull both employed MLS for user-data confidentiality and Biba for trusted computing base (TCB) protection. Composition must be predictable, deterministic, and ideally sensible.

Impose structures that facilitate assurance arguments. This can be done by separating policy and mechanism via a reference monitor and through well-defined KPI semantics (for example, locking).

Design for an increasingly concurrent kernel. Policies must not only behave correctly, but also scale with the features they protect.

Architecture of the MAC Framework. The MAC Framework, illustrated in Figure 2, is a thin layer linking kernel services, policies, and security-aware applications. Control passes from kernel consumers to framework to policies through roughly 250 entry points (object types × methods):

▸ *Kernel-service entry points* allow subsystems (for example, VFS) to engage the reference-monitor framework in relevant events and access control.

▸ *Policy entry points* connect the framework and policies, adding explicit label arguments relative to corresponding kernel-service entry points. They are supplemented by policy lifecycle events and library functions. Policies need implement only the entry points they require.

▸ Applications manage labels (on processes and files, among others) using the *label-management API*.

▸ *DTrace probes* allow entry-point tracing, profiling, and instrumentation.[8]

Collectively, these interfaces allow policies to augment kernel access control in a maintainable manner.

Entry-Point Invocation. To understand how these layers interact, let's follow a single file-write check through the kernel. Figure 3 illustrates `vn_write`, a VFS function implementing the `write` and `writev` system calls. The `mac_vnode_check_write` kernel service-entry point authorizes a write to a `vnode` (`vp`) by two subject credentials: `fp->f_cred`, which opened the file, and `active_cred`, which initiated the write operation. Policies can implement Unix *capability semantics* (`fp->f_cred`) or *revocation semantics* (`active_cred`). The `vnode` lock (`vp->v_lock`) is held over both check and use, protecting label state and preventing time-of-check-to-time-of-use race conditions.

Arguments excluded from entry points are as important as those included. For example, `vn_write`'s data pointer (`uio`) is omitted, as this data resides in user memory and cannot be accessed race-free with respect to the write. Similar design choices throughout the framework discourage behavior not safely expressible through the kernel synchronization model.

Wherever possible, it is best to take the perspective that kernel subsystems implement labeled objects, and that policies may be enforced through controls on method invocation. This approach is a natural fit for the kernel, which adopts an object-oriented structure despite an absence of language features in C. Once objects have been identified, placing entry points requires care: the more granular the KPI, the more expressive policies can be—at the cost of policy complexity. The fewer the calling sites, the easier they are to validate; too few, however, leads to inadequate protection. Entry-point design must also balance placing checks deep enough to allow insight into object types while minimizing enforcement points for a particular level of abstraction.

Figure 3. VFS invokes the MAC Framework to authorize file writes.

```
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 4. Framework access control on file writes; lock assertions and DTrace probes are central design elements.

```
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 5. Biba authorization of file writes.

```
#define LABEL(l) ((struct mac_biba *)mac_label_get((l), biba_slot))

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

Figure 4 illustrates `mac_vnode_check_write`, a thin shim that asserts locks, invokes interested policies, and fires a DTrace probe. Policies are not prohibited from directly accessing `vnode` fields; however, passing an explicit label reference avoids encoding `vnode` structure layout into policies in a common case, improving KPI and KBI resilience.

Policy entry-point invocation, encapsulated in `MAC_POLICY_CHECK`, is nontrivial: access to the policy list must be synchronized to prevent races with module unload, interested policies must be called, and results must be composed. The framework employs a simple composition metapolicy: if any policy returns failure, then access is denied. For example, an `EACCES` returned by Biba would be selected in preference to 0 (success) returned by MLS. The only exception lies in privilege extensions discussed later. This metapolicy is simple, deterministic, predictable by developers, and above all, useful.

Figure 5 illustrates Biba invocation: Biba checks its revocation configuration, unwraps policy-specific labels, and computes a decision using its dominance operator.

Kernel-Object Labeling. Many access-control policies label subjects and objects in order to support access-control decisions (for example, integrity or confidentiality levels). The MAC Framework provides policy-agnostic label facilities for kernel objects, label-management system calls, and persistent storage for file labels. Policies control label semantics—not only the bytes stored, but also the memory model: policies might store per-instance, reference-counted, or global data. For example, when a process creates a new socket, Biba propagates the current subject integrity level (for example, *low*) to the socket label. The partition policy, concerned with interprocess access control, labels only processes and not sockets, so will not assign a label value for the socket.

The framework represents label storage using `struct label`, which is opaque to both kernel services and policies. Where object types support metadata schemes (for example, `mbuf` tags that hold per-packet metadata), those are used; otherwise, label pointers are added to core structures (for example, `vnode`). Policies may borrow existing object locks to protect label data, where supported by the synchronization model.

From Research to Product

Having presented the design of the MAC Framework, let's turn our attention to policies found in FreeBSD-derived commercial or open source products. Table 1 and Figure 6 illustrate several such policy modules, their feature footprints, and ship dates. A number of factors contributed to the success of this transition:

The need for new access control was pressing. The classic Unix model failed to meet the needs of ISPs, firewalls, and smartphones. Simultaneously, the threat of attack became universal with ubiquitous networking and strong financial incentives for attackers.

Structural arguments for a framework were correct. Access-control extensibility is the preferred way of supporting security localization, catering to diverse requirements.

No one policy model has become dominant. Therefore, many must be supported.

Hardware performance improvement increased tolerance for security overhead. This was true even in consumer and embedded devices.

Table 1. Comparison of policies and their feature footprints.

Name	OSS	CP	Product	Type	Lab	Priv	Proc	VFS	IPC	Net	API	Sig
mac none	✓	-	FreeBSD	Null policy	-	-	-	-	-	-	-	-
mac_stub	✓	-	FreeBSD	Template policy	✓	✓	✓	✓	✓	✓	✓	-
mac_test	✓	-	FreeBSD	Framework self-test	✓	-	✓	✓	✓	✓	✓	-
mac_ugidfw	✓	✓	FreeBSD	File system firewall	-	-	-	✓	-	-	-	-
mac_biba	✓	✓	FreeBSD	Fixed integrity	✓	✓	✓	✓	✓	✓	✓	-
mac_lomac	✓	?	FreeBSD	Floating integrity	✓	✓	✓	✓	✓	✓	✓	-
mac_mls	✓	?	FreeBSD	Confidentiality	✓	-	✓	✓	✓	✓	✓	-
sebsd	✓	✓	FreeBSD	Type Enforcement	✓	✓	✓	✓	✓	✓	✓	-
sandbox	-	✓	Apple OS X	Rule-based	✓	✓	✓	✓	✓	-	✓	✓
quarantine	-	✓	Apple OS X	Taint-based	✓	-	-	✓	-	-	✓	-
tmsafetynet	-	✓	Apple OS X	Fixed integrity	✓	-	-	✓	-	-	✓	-
amfi	-	✓	Apple iOS	Fixed integrity	-	✓	✓	-	-	-	-	✓
sandbox	-	✓	Apple iOS	Rule-based	✓	✓	✓	✓	✓	-	✓	✓
mac_runasnonroot	-	✓	Apple iOS	Hardening	-	-	✓	-	-	-	-	✓
mac_pcap	-	✓	Juniper Junos	Grant BPF privs	✓	✓	-	✓	-	✓	✓	-
mac_veriexec	-	✓	Juniper Junos	Signed binaries	✓	-	-	✓	-	-	-	✓
sidewinder_te	-	✓	McAfee Sidewinder	Type Enforcement	✓	✓	✓	✓	✓	✓	✓	-
mac_ncircle	-	✓	nCircle IP360	Hardening	-	✓	-	✓	-	-	-	-

Key:
OSS: open source software
CP: shipped in a commercial product
Lab: uses subject or object label facility
Priv, Proc, VFS, IPC, Net: implements access-control entry points for privileges, processes, file system, interprocess communication, or the network stack
API: uses MAC Framework application APIs
Sig: provides or depends on application digital signatures

Open source technology transition works. FreeBSD provided not only a forum for collaborative research and development, but also a pipeline to commercial products.

The framework has evolved considerably since 2003 thanks to contributions from companies deploying it in products.

FreeBSD

FreeBSD is an open source operating system used to build online services, appliances, and embedded devices. FreeBSD or its components can be found in data centers (Internet Systems Consortium, Yahoo!), as a foundation for integrated products (NetApp and EMC Isilon storage appliances), and in embedded/mobile devices (Juniper switches and Apple iPhones). Its origins lie in BSD (the Berkeley Software Distribution), developed in the 1970s and 1980s.[12] BSD originated a number of central Unix technologies, including FFS (the Fast File System) and the Berkeley TCP/IP stack and sockets API. The BSD license and its variations (MIT, CMU, ISC, Apache) have encouraged technology transition by allowing unrestricted commercial use. FreeBSD's diverse consumers both motivate and are the perfect target for security localization.

The MAC Framework is a complex piece of software; although the framework itself is only 8,500 lines of code, with 15,000 lines in reference policies, it integrates with a multimillion-line kernel. The transition to production relied on several factors, including increasing confidence in mediation and response to community feedback on design, compatibility, and performance. The framework, as first shipped in FreeBSD 5.0, was marked as *experimental*, with several implications:

- Enabling it required recompiling the kernel.
- Documentation marked it as potentially incomplete, unstable, or insecure, and therefore unsupported.
- Programming and binary interface (API, KPI, ABI, and KBI) stability was disclaimed, allowing change without formal depreciation.

Merging the framework while still experimental was key to gaining users who could help validate and improve the approach, while retaining the flexibility to make changes. Two concerns needed to be addressed before the framework could be considered production worthy:

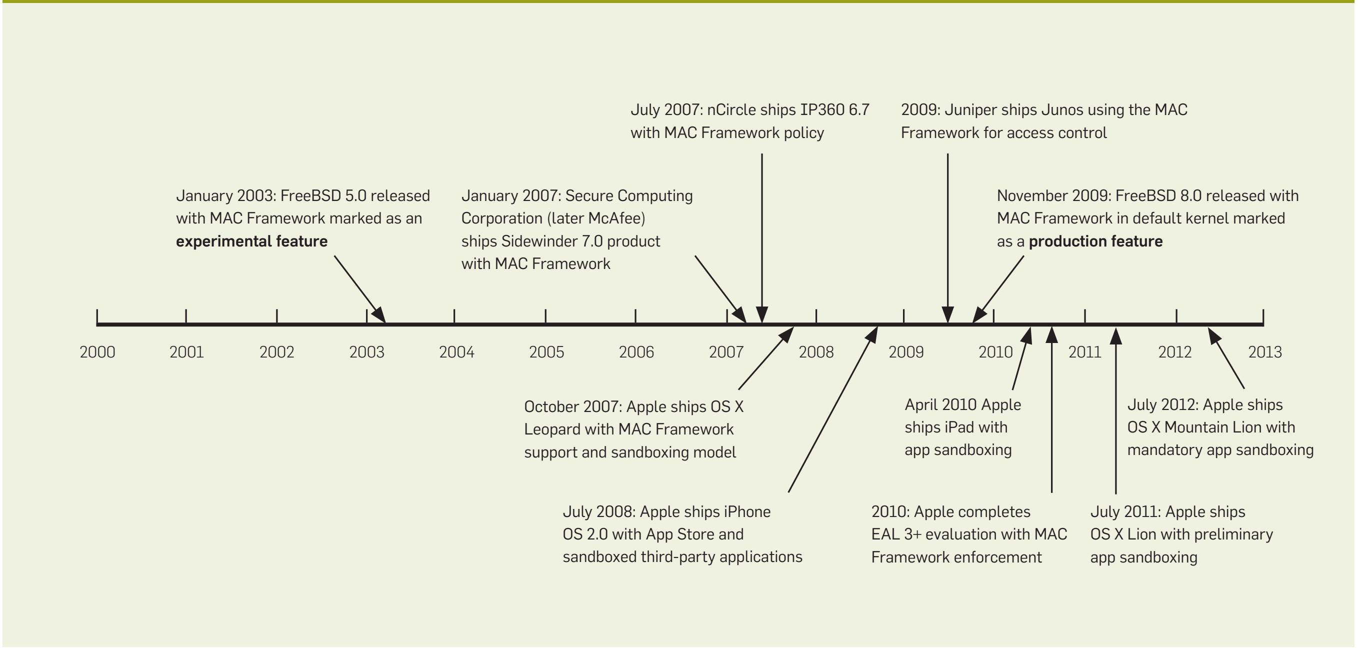
- Binary compatibility impact for the kernel, policies, and other modules must be better understood.
- Performance must be analyzed and optimized based on community review.

KPI and KBI Resilience. FreeBSD policy dictates that certain classes of kernel modules compiled against a release must work with later minor versions in the same series (for example, a FreeBSD 9.0 network device driver should work with FreeBSD 9.1). The goals were to avoid disrupting the KBIs of consumer subsystems and to offer similar levels of binary compatibility for policy modules. Label storage opacity for subsystems and policies was the primary area of refinement, which avoids encoding kernel data-structure internals into policies if they require only label access, as well as providing flexibility to change label implementation.

Performance Optimization. Many FreeBSD deployments are extremely performance sensitive, requiring minimal overhead, especially if the framework is disabled. As sites select policies based on local security-performance trade-offs, it is also desirable for policies to incur only the performance penalties of features they actually use—*performance proportionality*. As shipped in FreeBSD 5.0, however, regressions were measurable, an obstacle to enabling the framework by default.

Label Allocation Trade-offs. Even when the framework was compiled out, bloat from adding a label to kernel data structures (especially packet `mbufs`) created significant allocation-time zeroing cost. In FreeBSD 5.1, inlined `mbuf` labels were replaced with pointers, and for all object types in 5.2; this decreased costs for non-MAC

Figure 6. Timelines of selected MAC Framework-based product ship dates.



July 2007: nCircle ships IP360 6.7 with MAC Framework policy
2009: Juniper ships Junos using the MAC Framework for access control
January 2003: FreeBSD 5.0 released with MAC Framework marked as an experimental feature
January 2007: Secure Computing Corporation (later McAfee) ships Sidewinder 7.0 product with MAC Framework
November 2009: FreeBSD 8.0 released with MAC Framework in default kernel marked as a production feature
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
October 2007: Apple ships OS X Leopard with MAC Framework support and sandboxing model
April 2010 Apple ships iPad with app sandboxing
July 2012: Apple ships OS X Mountain Lion with mandatory app sandboxing
July 2008: Apple ships iPhone OS 2.0 with App Store and sandboxed third-party applications
2010: Apple completes EAL 3+ evaluation with MAC Framework enforcement
July 2011: Apple ships OS X Lion with preliminary app sandboxing

kernels at the expense of additional allocation and indirection for MAC-enabled kernels.

Label allocation was even more measurable with the framework enabled—and unnecessary for unlabeled policies. The effect was most pronounced with network packets and led, in FreeBSD 5.1, to a per-policy flag to request packet labels. In 8.0, this approach was generalized so that labels were allocated only for object types for which at least one loaded policy defined an initialization entry point. This effectively eliminated the cost of labeling when not required by a policy, restoring performance proportionality and satisfying the general case well. However, one commercial product that used packet labeling, the McAfee Sidewinder Firewall, saw sufficient overhead to bypass the label abstraction in favor of direct structure modification.

Minimizing Synchronization Overheads. With the framework compiled in, lock-protected reference count operations on entry-point invocation were easily measurable for frequent operations, such as per-packet delivery checks. As multicore hardware became more common, lock (and later cache-line) contention also became significant.

Beginning in FreeBSD 5.2, policies were divided into *static* and *dynamic* sets to help fixed-configuration embedded systems. The former were compiled in or loaded at boot and unloadable thereafter, and hence required no synchronization. Dynamic policies—those loaded after boot, or potentially unloadable—still required multiple lock operations.

In FreeBSD 8.0, synchronization was further optimized so that the MAC Framework could be shipped in the default kernel. This effort benefited from continuing improvements in kernel scalability driven by increasingly common eight-core machines. Particularly critical were *read-mostly locks*, which do not trigger cache-line migrations during read-only acquisition, at the cost of more expensive exclusive acquisition—perfect for infrequently changed policy lists.

It is desirable for policies to incur only the performance penalties of features they actually use—performance proportionality.

nCircle IP360 Appliance

nCircle Network Security produces a FreeBSD-based appliance, the IP360, to scan networks for vulnerable software and Sarbanes-Oxley compliance. While most of its security requirements could be captured with conventional DAC, customers requested the ability to audit appliance content and configuration directly. To meet this requirement, while limiting potential damage in case audit access is misused or compromised, nCircle developed a custom policy.

The policy authorizes an *audit user* to read all file-system and configuration data, bypassing permissions, while also preventing file-system writes. The MAC Framework could express only a subset of this augmentation: policies could constrain rights but not grant them. nCircle therefore enhanced the framework to allow control over fine-grained *system privileges*.

Privilege Extensions. Operating-system privilege confers the right to bypass operating-system security policies (for example, changing system settings or overriding DAC or the process model). In classic Unix, system privileges are granted to any process running as the root user. To meet nCircle's goals, a policy must be able to augment the kernel's default privilege policy to grant (and moderate) privileges for other users. This presented two technical challenges: how to identify and distinguish different types of privilege; and how to add extensibility to the existing privilege model. These problems resemble, in microcosm, the larger concern addressed by the MAC Framework—structuring of a reference monitor for extensibility—and seemed a natural fit despite a departure from the original design choice to only limit, rather than grant, rights.

All existing kernel privilege checks were analyzed and replaced with checks for specific named privileges. Privilege checking was then reworked to include an explicit composition policy for sources and limitations of privilege, including two new MAC Framework entry points: `mac_priv_check` follows the standard entry-point conventions, accepting a credential, named privilege arguments, and restricting privileges by returning an error; `mac_priv_grant` diverges from this model by overriding the base operating-system policy to grant new rights, using a new composition operator that allows any policy to grant a right, rather

Table 2. Apple OS X applications may use one of several statically configured profiles, or define their own.

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 temporary folders
kSBXProfilePureComputation	Only Mach IPC to the host process is permitted

than requiring them all to agree.

Existing policies were updated to take advantage of the new features, providing stronger nondiscretionary control of the root user. For example, the Biba policy now limits access to a number of privileges that might allow bypass of the process model or system reconfiguration when operating as the root user without Biba privilege. These features shipped in FreeBSD 7.0.

The nCircle MAC Policy. The nCircle policy extends (and restricts) rights available to the *audit user*:

- It identifies a specific user ID to which all remaining policy activities apply.
- Privileges are granted, including read access to the kernel log and firewall configuration, and file read/lookup protections are overridden.
- VFS entry points deny write access to all objects and read access to certain files such as the password file.

With these enhancements, the nCircle policy is able to combine controlled privilege escalation with mandatory constraints, meeting product needs while minimizing local operating-system modification.

Juniper Junos

The Junos router operating system runs on the control planes of all Juniper routers and switches. Juniper maintains substantial local modifications to FreeBSD and is undergoing a multiyear process to minimize its patches by returning improvements to the FreeBSD community and increasing use of operating-system extensibility frameworks that allow local features to be cleanly grafted onto an unmodified operating system. As part of that project, Juniper has been moving local security extensions into MAC Framework policies, both to reduce conflicts during FreeBSD updates and to prepare certain policies for upstreaming. Junos ships with four local security extensions:

- `mac_runasnonroot`. Ensures that third-party applications written against the Junos SDK are not run as the root user.
- `mac_pcap`. Allows Junos SDK applications to capture packets despite not running as root.
- `mac_veriexec`. Implements support for digitally signed binaries.
- Junos SDK sandboxing. Constrains third-party applications based on `mac_veriexec certificates`.

The `mac_runasnonroot` and `mac_pcap` extensions first shipped as framework policies in 2009. Then `mac_veriexec` shipped in 2012, replacing a previous directly patched implementation. Juniper is preparing to migrate Junos SDK sandboxing to the MAC Framework to reduce local patches further, as well as upstream `mac_veriexec`.

These policies required minor changes to the MAC Framework, including additional entry points; perhaps most interesting is a new `O_VERIFY` flag to the `open` system call, which signals to the framework that the userspace runtime linker has requested that a file be validated.

Apple OS X and iOS

In quick succession, Apple released versions of OS X Leopard for the desktop/server in 2007, and iPhone OS 2 for the iPhone and iPod Touch in 2008, incorporating the MAC Framework as a reference-monitor framework. OS X Snow Leopard shipped with three MAC policies:

- **Sandbox.** Provides policy-driven sandboxing of risky components that process untrustworthy data such as network services and video codecs.
- **Quarantine.** Taints downloaded files, supporting a user dialog displaying the originating website.
- **Time Machine Safety Net.** Protects the integrity of Time Machine backups.

With OS X Mountain Lion, applications distributed via Apple's App Store have mandatory sandboxing. Apple's iOS 2.0 shipped with two policies: Sandbox and one additional:

- **Apple Mobile File Integrity (AMFI).** Works in concert with a code-signing facility, terminating apps whose digital signatures have been invalidated at runtime; exempts debugging during app development.

Collectively the policies support system integrity and provide strong separation between apps in order to keep data private. Both OS X and iOS diverge substantially from our design expectations for the MAC Framework, requiring significant adaptation.

XNU Prototype. Apple began beta testing OS X in 2000, and the promise of a commodity desktop operating system with an open source kernel was difficult to ignore. The XNU kernel is a sophisticated blend of Carnegie Mellon University's Mach microkernel, FreeBSD 5.0, cherry-picked newer FreeBSD elements, and numerous features developed by Apple. With these foundations, it seemed likely that the MAC Framework approach, and even code, would be reusable.

Though not a microkernel, XNU (short for X is not Unix) adopts many elements from Mach, including its scheduler, interprocess communication (IPC) model, and VM system. The FreeBSD process model, IPC, network stack, and VFS are grafted onto Mach, providing a rich POSIX programming model. Apple-developed kernel components in the first release of OS X included the I/O Kit device-driver framework, network kernel extensions (NKEs), and the HFS+ file system; this list has only grown over time.

Interesting questions abounded: for example, would ideas developed in the DTMach[16] and DTOS[17] microkernel projects apply better or worse than the monolithic kernel approach in the MAC Framework? Between 2003 and 2007, the increasingly mature MAC Framework was ported to OS X.[18]

Adapting to OS X. The MAC Framework required a detailed analysis of the FreeBSD kernel and is tightly integrated with low-level memory management and synchronization, as well as higher-level services such as the file system, IPC, and network stack. While the adaptation to OS X was able to rely heavily on Apple's use of FreeBSD components, fundamental changes were needed to reflect differences between FreeBSD and XNU.

The first step was integrating the MAC Framework with the closely aligned BSD process model, file system, and network stack. High-level architectural alignment made some of the adaptation easy, but nontrivial differences were also encountered. For example, FreeBSD's Unix file system (UFS) considers directories to be specialized file objects, whereas HFS+ considers the directory and object attribute structure, or *disk catalog*, to be a first-class object. This required changes to both the framework and XNU.

Next, coverage was extended to include Mach tasks and IPC. Each XNU process links a Mach task (scheduling, VM) with a FreeBSD process (credentials, file descriptors), presenting a philosophical problem: is the MAC Framework part of Mach or BSD? While useful architecturally, the Mach-BSD boundary in XNU proves artificial: references frequently span layers, requiring the MAC Framework to serve both. Label modifications on BSD process labels are mirrored to corresponding Mach task labels.

Mach ports are another case in which microkernel origins come into conflict with the monolithic kernel premise of the MAC Framework. Unlike BSD IPC objects, with kernel-managed namespaces, Mach ports rely on userspace namespaces managed by `launchd` (for example, for desktop IPC). Taking a leaf from DTOS, `launchd` is responsible for labeling and enforcement but queries the reference monitor to authorize lookups. A userspace *label handle* abstraction similar to the kernel `label` structure serves this purpose.

Adoption by Apple. Apple is the world's largest vendor of desktop Unix systems and was among the first to deploy Unix in a smartphone. It has likewise seen exploding use cases and new security requirements motivated by ubiquitous networking and malicious attackers. Apple's adoption of the MAC Framework was not assured, however, as competing technologies were also considered, motivated by similar observations, awareness of future product directions, performance concerns, and our research.

Alternatives included system-call interposition-based technology similar to that discussed earlier, and Apple's Kauth[3] (short for kernel authorization), an authorization framework targeted at antivirus vendors (modeled in part on the MAC Framework). Apple found arguments about the fallibility of system-call interposition convincing, and in the end adopted two technologies: Kauth for third-party antivirus vendors; and the more expressive and capable MAC Framework for its own sandboxing technologies.

The Sandbox Policy. Since Apple's OS X and iOS policy modules are not open source, we are unable to consider their implementations, but public documentation exists for the Sandbox policy used by Mac OS X components and third-party applications such as Google's Chrome Web browser. Sandbox allows applications voluntarily to restrict their access to resources (for example, the file system, IPC namespaces, and networking). Process sandbox profiles are stored in process labels.

Bytecode-compiled policies can be set via public APIs, or by the `sandbox-exec` helper program. Applications may select from several Apple-defined policies (Table 2) or define custom policies. Several applications use default policies such as the iChat video codec, which employs the computation-only profile limited to IPC with the host process. Many other software components, such as Spotlight indexing, the BIND name server, Quicklook document previews, and the System Log Daemon, utilize custom profiles to limit the effects of potential vulnerabilities.

Figure 7 shows excerpts from the `common.sb` profile used by Chrome, illustrating key Sandbox constructs: coarse controls for `sysctl` kernel-management interfaces and shared memory, and fine-grained regular expression matching of file paths. File path-based control is a highlight of the Sandbox policy, addressing programmer models much better than file labels in Biba, MLS, and TE. Path-based schemes are difficult to implement on the Unix VFS model, which considers paths to be second-class constructs. Whereas FreeBSD permits files to have zero (unlinked

Figure 7. Chrome OS X sandbox policy excerpts.

```
(deny default)

; 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)

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

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

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

but open), one, or multiple names (hard links), HFS+ implements a parent pointer for files and ensures that the name cache always contains the information required to calculate unambiguous paths for in-use files.

While Sandbox is used with many OS X services, a number of third-party applications incorporate strong assumptions of *ambient authority*, the ability to access any object in the system. With the iPhone, Apple broke this assumption: applications execute in isolation from system services and each other. This model is now appearing in OS X and could similarly help protect device integrity against misbehaving apps and, increasingly, end-user data.

Performance Optimizations. OS X and iOS were shipped with the MAC Framework prior to FreeBSD 8.0's performance optimizations, requiring Apple to make its own optimizations based on product-specific constraints. As with FreeBSD optimizations, these were generally concerned with the overhead of framework entry and labeling. By default, labeling is compiled out of the kernel for certain object types; for others, such as `vnodes`, policies may selectively request label allocation, catering to the often-sparse labeling use in OS X's policies.

In FreeBSD, framework instrumentation and synchronization optimizations rely on all-or-nothing distinctions between sites willing to pay for additional access-control extension. In OS X, the assumption is that sandboxing is used on most machines, but selectively applied to high-risk processes. To this end, each process carries a mask, set by policies, indicating which object types require enforcement. As OS X adopts more universal sandboxing, as is the case in iOS, it may be desirable to apply more global optimizations as in FreeBSD.

Reflections

Over the past decade, the MAC Framework has become the foundation for numerous instances of security localization, allowing local access-control policies to be composed with the still-popular Unix discretionary access control (DAC) model—a timely convergence of industry requirements and research. Deploying via open source proved a successful strategy, providing a forum for collaborative refinement, access to early adopters, and a path to numerous products.

The MAC Framework has become the foundation for numerous instances of security localization, allowing local access-control policies to be composed with the still-popular Unix discretionary access control model.

Perhaps the most surprising adoption was at McAfee itself: when the framework was open sourced by McAfee Research, Secure Computing Corporation (then a competitor) adopted it for Sidewinder, which McAfee later acquired. More generally, this speaks to the success of open source in providing a venue in which competing companies can collaborate to develop common infrastructure technologies. The industry's adoption of open source foundations for appliances and embedded devices has been well-catered to by our access-control extensibility argument:

- Security localization in devices has become widespread.
- The criticality of multiprocessing has only increased.
- Security label abstractions have proven beneficial beyond their MAC roots.
- Non-consensus on access-control policies continues.

The MAC Framework, however, also required refinement and extension to address several unanticipated concerns:

- The desire to revisit the structure of Unix privilege.
- The importance of digital signatures when applying access control to third-party applications.
- Continued tensions over the desire for name-based vs. label-based access control.

New Design Principles. In light of extensive field experience with the MAC Framework, we have added several new design principles:

Policy authors determine their own performance, functionality, and assurance trade-offs. Policies may not require heavyweight infrastructure (for example, labels), so offer performance proportionality.

Traceability is a key design concern.

Programming and binary interface stability is critical. API, ABI, KPI, and KBI sustainability is often overlooked in research, where prototypes are frequently one-offs without multi-decade support obligations.

Manipulating operating-system privilege is important to policies that augment rather than supplement DAC.

It is clear from the work of downstream consumers, however, that two further principles are now also necessary:

Application authors are first-class principals. Apple's App Store and Juniper's SDK both employ application signatures and certificates as policy inputs.

Applications themselves require flexible access control to support application compartmentalization.

This latter observation led us to develop the application-focused Capsicum protection model,[21] recently shipped as an experimental feature in FreeBSD 9.0. It can be viewed as complementary to policy-driven kernel access control.

Domain-Specific Policy Models. Why no consensus has been reached in the expression of operating-system policies is an interesting question—certainly, proponents of successive policy models have argued that their models capture the key concerns in system design. In catering to a variety of models, our observations are twofold: first, policy models aim to capture aspects of the *principle of least privilege*[15] but often in fundamentally different forms (for example, information flow vs. system privileges), making their approaches complementary; second, different models address different spaces in a multidimensional trade-off between types of expression, assurance, performance, administrative complexity, implementation complexity, compatibility, and maintainability. This instead reflects a consensus for *domain-specific policy models*.

The Value of Extensibility. Does the need for significant design enhancement confirm or reject the hypothesis of access-control extensibility? Further comparison to similar frameworks, such as VFS and device drivers, seems appropriate: both are regularly extended to adapt to new requirements such as changes in distributed file-system technology or improvement in power management. The willingness of industrial consumers to extend the framework and return improvements reflects our fundamentally economic hypothesis regarding extensibility: managing the upstream-downstream relationship for significant source-code bases is a strong motivator. Widespread and continuing deployment of the MAC Framework appears to confirm the more general argument that access-control extensibility is a critical aspect of contemporary operating-system design.

Acknowledgments

Systems research emphasizes the practical application of ideas to real-world systems: only by implementing an idea can you fully understand it; this is even more true in the transition from research to practice. Acknowledgments are due to a large cast spanning many institutions; to view full list, see queue.acm.org. Helpful feedback on this article came from Ross Anderson, Simon Cooper, Jon Crowcroft, Simon Gerraty, Matthew Grosvenor, Steve Hand, Mark Handley, Steve Kiernan, Anil Madhavapeddy, Peter G. Neumann, George Neville-Neil, and Mike Silbersack.

This work was supported by DARPA/AFRL contract FA8750-10-C-0237 (CTSRD), with previous support from the DARPA CBOSS and SPAWAR SEFOS contracts, spanning the DARPA CHATS and DARPA CRASH research programs. The views, opinions, and/or findings in this article are author's and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA, the U.S. Navy, AFRL, or the Department of Defense. Google, Inc. also supported this work.

References

1. Abrams, M.D., Eggers, K.W., LaPadula, L.J. and Olson, I.M. A generalized framework for access control: An informal description. In *Proceedings of the 13th NIST-NCSC National Computer Security Conference* (1990), 135–143.
2. Anderson, J.P. Computer Security Technology Planning Study. Technical report, Electronic Systems Division, Air Force Systems Command, 1972.
3. Apple Inc. Kernel authorization. Technical Note TN2127, 2007; http://developer.apple.com/technotes/tn2005/tn2127.html.
4. Badger, L., Sterne, D.F., Sherman, D. ., Walker, K.M. and Haghighat, S.A. Practical domain and type enforcement for Unix. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy 66* (1995). IEEE Computer Society.
5. Bell, D.E., and L.J. LaPadula. Secure computer systems: mathematical foundations and model. Technical Report M74-244. Mitre Corp., Bedford, MA, 1973.
6. Biba, K. Integrity considerations for secure computer systems. Technical Report TR-3153. Mitre Corp., Bedford, MA, 1977.
7. Boebert, W.E. and Kain, R.Y. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.
8. Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H. Dynamic instrumentation of production systems. In *Proceedings of the Usenix Annual Technical Conference* (Berkeley, CA, 2004). Usenix Association.
9. Fraser, T., Badger, L. and Feldman, M. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.
10. Kleiman, S.R. Vnodes: An architecture for multiple file system types in Sun Unix. In *Proceedings of the Summer 1986 Usenix Conference*.
11. Loscocco, P.A. and Smalley, S.D. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 Usenix Annual Technical Conference*. Usenix Association, 29–42.
12. McKusick, M.K., Neville-Neil, G.V. *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, 2004.
13. Neumann, P.G., Boyer, R.S., Feiertag, R.J., Levitt, K.N. and Robinson, L. A provably secure operating system: the system, its applications, and proofs, second edition. Technical Report CSL-116. Computer Science Laboratory, SRI International, 1980.
14. Ott, A. Rule-set-based access control (RSBAC) for Linux (2010); http://www.rsbac.org/.
15. Saltzer, J.H. and Schroeder, M.D. The protection of information in computer systems. In *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.
16. Sebes, E.J. Overview of the architecture of Distributed Trusted Mach. In *Proceedings of the Usenix Mach Symposium* (1991). Usenix Association, 20–22.
17. Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. and Lepreau, J. 1999. The Flask security architecture: system support for diverse security policies. In *Proceedings of the 8th Usenix Security Symposium* (1999). Usenix Association, 123–139.
18. Vance, C., Miller, T. C., Dekelbaum, R., Reisse, A. 2007. Security-enhanced Darwin: Porting SELinux to Mac OS X. In *Proceedings from the Third Annual Security Enhanced Linux Symposium* (2007).
19. Watson, R.N.M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First Usenix Workshop on Offensive Technologies*. Usenix Association, 2007, 1–8.
20. Watson, R.N.M. New approaches to operating system security extensibility. Technical Report UCAM-CL-TR-818. University of Cambridge, Computer Laboratory, 2012.
21. Watson, R.N.M., Anderson, J., Laurie, B. and Kennaway, K. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th Usenix Security Symposium* (2010). Usenix Association.
22. Watson, R.N.M. Feldman, B., Migus, A. and Vance, C. Design and implementation of the TrustedBSD MAC Framework. In *Proceedings of the Third DARPA Information Survivability Conference and Exhibition* (2003). IEEE.
23. Wright, C., Cowan, C., Morris, J., Smalley, S. and Kroah-Hartman, G. 2002. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th Usenix Security Symposium* (2002). Usenix Association.

Robert N.M. Watson is a senior research associate at the University of Cambridge Computer Laboratory, and a Research Fellow at St John's College Cambridge. He was a senior principal scientist at SPARTA ISSO, and a senior research scientist at McAfee Research, where he led the development of a kernel access control extension framework for the open source FreeBSD operating system, now used in products such as Junos, Apple OS X, and iOS.