The Lazy Kernel Hacker and Application Programmer

W. Michael Petullo Jon A. Solworth
University of Illinois at Chicago
mike@flyn.org, solworth@rites.uic.edu

Abstract

Over time, programming languages have become more abstract, adding features such type safety, garbage collection, and improved modularization. These improvements contribute to program quality and have allowed application programmers to write increasingly complicated programs. There are thousands of programming languages in existence, and features are routinely drawn from the existing set and recombined to produce the next generation.

Like programming languages, operating systems fundamentally affect the quality of the software (including language runtimes) written on them. After all, the operating system provides—primarily through system calls—the interface programs must use to interact with users, other programs, and other networked computers. But operating systems have evolved more slowly than programming languages, due in no small part to the difficulty of writing them. Here we describe the interface embodied by our operating system, Ethos, and how we used virtualization and other techniques to develop the operating system more quickly.

1. Introduction

For decades, UNIX (and POSIX, in general) has provided a dominant Operating System (OS) interface, presently providing the basis for servers, workstations, tablets, and cellphones. Although UNIX has accreted functionality over time (e.g., networking, graphical environments, and sensor support), its system call interface has not undergone a clean-slate redesign [34]. This contrasts with programming languages, which from the 1970s until today have radically changed. Modern languages provide features such as type safety, garbage collection, and improved modularization. These abstractions relieve the application programmer from focusing on low-level details, enabling him to write higher-quality, more complex applications.

We became interested in the level of abstraction found in OS system calls after accumulating evidence that current levels are detrimental to security. For example, connect and accept on POSIX leave encryption and user authentication to the application[1]. Even well-meaning developers routinely misuse or avoid the various application-level network security toolkits necessary to provide missing protections, and this results in software with security vulnerabilities [13, 17, 35, 45]. Thus we began work on Ethos, an operating system that addresses security by providing system calls at a higher level of abstraction. Ethos system calls provide protections that cannot be bypassed. When coming up with a strategy for building Ethos, we wanted to quickly build prototypes that would allow us to experiment with these interfaces.

Roscoe et al. used the term *disruptive virtualization* [37] when describing how virtualization could allow more expedient experimentation with OS interfaces, and we tailored some of their ideas for our own use. Writing an OS traditionally is a difficult task. This is due to the unforgiving environment of communicating directly with hardware, the idiosyncrasies found in computer architectures, the requirement for writing device drivers, and the need to complete many subsystems before the OS becomes useful. Here we discuss how we used virtualization to more expediently accomplish each of these requirements.

[1] With IPsec, POSIX can provide encryption independent of application code, but it is non-trivial for an application to determine if IPsec is active, and it is still up to the application programmer to make such checks.

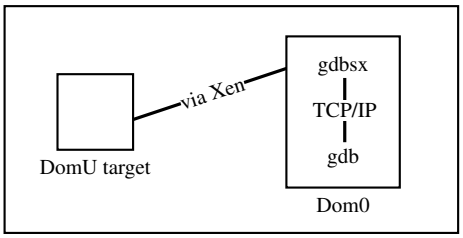


Figure 1: The Xen kernel debugging architecture

2. The appeal of a VMM-based approach

We began work on Ethos before there was stable Hardware Virtual Machine (HVM) support in Xen [2]. Because of this, we chose not to build Ethos on top of an existing microkernel. Instead, we built Ethos by adding user-space processes, system calls, etc. to Xen's MiniOS. There are many known advantages to targeting a Virtual Machine Monitor (VMM) when developing a new OS. This includes debugging support, profiling support, device support, and providing for backwards compatibility. We summarize each of these here before describing some additional tricks in §3.

Debugging The Linux kernel provides a debugging interface made up of gdb and a kernel component, kgdb. The kgdb stub consumes 5,802 Lines of Code (LoC) on Linux 3.2. Furthermore, using kgdb requires a second computer, connected to the debug target using a serial cable. Because Ethos targets Xen, it is unnecessary for us to implement special debugging support code in our kernel. Instead, we take advantage of gdbsx, a feature provided by Xen.

Figure 1 summarizes the gdbsx architecture. gdbsx is a program that runs on Dom0 that (1) communicates via Xen with the target virtual machine, and (2) communicates with gdb over a TCP/IP socket. The only special requirement from Ethos when using gdbsx is that Ethos must have been compiled into an executable with debugging information.

To automate the use of gdbsx, we wrote a program named ethosDebug that runs Ethos, gdbsx, and gdb. ethosDebug presents the user with four windows which provide Ethos' console output, terminal access to Ethos, other utility output, and the gdb interface.

Profiling Profiling also benefits from the use of Xen. Xen provides a profiler, Xenoprof [29]. Like gdbsx, using Xenoprof to profile an Ethos kernel requires only that the kernel contain debugging information[2].

We use Xenoprof in conjunction with xentop, a tool that displays various statistics about running Xen domains. Figure 2 lists the partial output of a profile of two Ethos domains. Domain 374 (column 4) is a server and domain 376 is a client, and the two domains are performing a networking benchmark. We had used Xenoprof to help tune portions of the kernel to maximize network speed, and the output indicates success because the work is CPU-bound due to two cryptographic functions.

Device drivers Device drivers consume a significant number of LoC in OSs. We used the utility cloc to count 5,625,090 lines of device driver code in Linux 3.2, around 50% of the total LoC in the kernel. Device driver code is often a source of bugs, including security vulnerabilities. One study found that they are a source of three to seven times more errors than general kernel code [8]. Reasons for this high bug rate include malfunctioning devices and poor documentation. Despite the large effort required, writing device drivers is often uninteresting from a research point of view.

Luckily, virtualization provides a solution. By targeting Xen, Ethos need only implement drivers for Xen's limited set of virtual devices. Ethos' network device driver is 462 LoC, and its console driver is 296 LoC. Although Ethos implements a single paravirtualized driver per device class, it is able to make use of any physical device supported by Xen Dom0.

Backward compatibility Perhaps most obviously, virtualization helps address the problem of backwards compatibility. Developing an OS that provides clean-slate interfaces implies that existing applications are not easily ported. This has been described as the *application trap*. However, the use of Ethos is worthwhile—even for a single application—because its use does not preclude the use of other OSs supporting legacy applications on the same computer. This has an obvious cost advantage—it is not necessary to purchase two computers for each Ethos developer that wants to run Linux and Ethos simultaneously. Here we find encouragement for specialized OSs in general.

3. Laziness in Ethos

We make tongue-in-cheek use of the word *lazy* to describe focusing time spent while developing a new OS to maximize interesting research opportunities. Our use of lazy is similar to Larry Wall's [46]. Our focus on developing Ethos is on providing new interfaces, and so we want to work quickly through the more mundane portions of OS construction. To make progress, we chose to take shortcuts when developing Ethos' networking stack and filesystem. In addition, we tried to maximize good software engineering through the use of something we call *mixins* and deliberate testing.

Shadowæmon Many of the techniques we describe below use shadowæmon to accelerate prototyping. Shadowæmon is a Linux program that provides various services to an Ethos kernel running in another Xen domain. Communication between an Ethos kernel and shadowæmon takes place over Xen's virtual network using a series of Remote Procedure Calls (RPCs), listed in Table 1. (We implemented Ethos-to-Ethos networking as a set of RPCs too, so this required little additional labor.) While we expect to eventually replace shadowæmon's services with native Ethos implementations, shadowæmon allowed us to more quickly implement and test Ethos' design.

[2] The support for this passive profiling has not been integrated into the mainline Linux kernel, so we occasionally port HP's patch to newer kernels and make this work available at `http://www.ethos-os.org/xenoprof`.

Name	Description
Ping	A connectivity test
GetUsers	Get the system user accounts
MetaGet	Get the metadata associated with a filesystem node
FileRead	Read the contents of a named file
FileWrite	Write an object to a named file
DirectoryRemove	Remove the named directory
DirectoryCreate	Create the named directory
Random	Generate random data

Table 1: List of RPCs supported by shadowæmon

Networking As with most modern OSs, support for networking in Ethos is necessary. For Ethos, networking is required because we are interested in studying secure network interfaces. An Ethos host maintains two network interfaces, one to communicate with the network, and another to communicate with shadowæmon. When Ethos boots, it reads a MAC and IP address for each of these interfaces from the command line provided by Xen. However, we were not interested in spending the time to implement the components of networking that are not interesting to our research goals. This includes Address Resolution Protocol (ARP) and a host routing table.

Linux 3.2's arp.c contains 1,000 LoC. Instead of implementing ARP and a host routing table, we engineered our Xen configuration so that both would be unnecessary in Ethos. First, we observed that if we configured Xen to route packets between its virtual and physical interfaces (instead of bridging them), then Ethos could use the same MAC address (i.e., Dom0's) for every outgoing packet (we depict this in Figure 3). This is because in such a configuration, every packet is either destined for the Dom0 host or will be routed by the Dom0 host. The MAC Dom0 uses for its virtual network devices is known: fe:ff:ff:ff:ff:ff.

Incoming packets are slightly more complicated as is shown in Figure 4. Xen's vif-route script updates Dom0's routing tables for each new Xen domain: the script inserts a point-to-point route (Figure 4b) for each Ethos domain going out Dom0's virtual interface (e.g., vifX.Y). But each host and router on a subnet must also discover Ethos' MAC—normally hosts obtain this information using ARP. Luckily, Linux has a feature called proxy ARP which is useful here.

We configure each Dom0 virtual network interface and Dom0's primary physical network interface with the same IP address. This partitions a single subnet so that each virtual network interface on Dom0 connects to a single Ethos host, and the physical interface connects to the remainder of the subnet. We configure Dom0 to forward packets between each partition and turn on proxy ARP (sysctl flag net.ipv4.conf.all.proxy_arp). With this configuration, Dom0 will answer ARP requests on behalf of each Ethos host assuming (1) the request was received on interface n's partition, and (2) the target address belongs to a host that exists on an interface other than n. The final step is to ensure Dom0 has ARP table entries for each Ethos host. Ethos immediately sends a packet to shadowæmon upon booting, and shadowæmon uses this packet to update Dom0's static ARP table (Figure 4c). Thus when Dom0 receives a packet destined to an Ethos host, its routing/ARP tables allow it to deliver the packet correctly.

Filesystem Ritchie and Thompson stated "*The most important job of* UNIX *is to provide a file system*" [36]. It is very likely that even research OSs need one. Unfortunately, filesystems are difficult to develop—an unappealing prospect unless you are a filesystem researcher. We counted 25,829 lines in Linux 3.2's ext4 directory and 51,584 in btrfs. Previous research has shown that even production

```
CPU: AMD64 family15h, speed 4551.44 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00...
samples  %        image name               app name                 symbol name
1235892  40.9540  ethos.x86_32.elf         domain374-kernel         ._morebits
739271   24.4974  ethos.x86_32.elf         domain376-kernel         ._morebits
96368     3.1934  ethos.x86_32.elf         domain374-kernel         crypto_scalarmult_curve25519...
57106     1.8923  ethos.x86_32.elf         domain376-kernel         crypto_scalarmult_curve25519...
...
```

Figure 2: Profile of running Ethos kernel

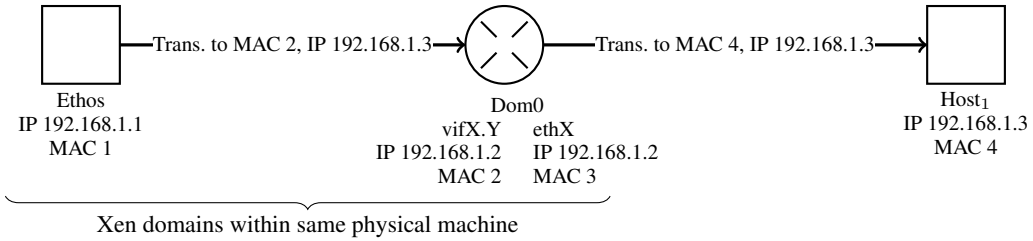
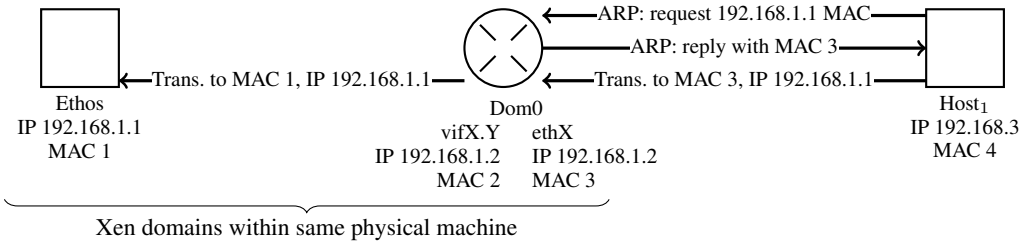


Figure 3: Sequence diagram: Ethos sends out a packet



(a) Sequence diagram

IP address	Interface
192.168.1.1	vifX.Y
192.168.1.0	eth0

(b) Routing Table

IP address	MAC
192.168.1.1	1

(c) Manual ARP Table

Figure 4: Using proxy ARP to deliver an incoming packet to Ethos

filesystems contain many bugs [38]. We have reduced this effort to shadowæmon's 814 LoC and an Ethos component of 1,754 LoC.

Instead of writing a filesystem for our Ethos prototype, we decided to take advantage of the existing filesystem on Dom0. When an Ethos application invokes a write system call, Ethos sends shadowæmon a FileWrite RPC that contains a path and the contents to write. Shadowæmon then writes the file to its local filesystem. A similar process supports an application's use of the read, fileInformation, removeFile, createDirectory, and removeDirectory system calls. The fileInformation system call is interesting in that Ethos supports file metadata typically not present on Linux. Here shadowæmon makes use of Linux's getxattr/setxattr system calls to store Ethos metadata along with the files it describes. Shadowæmon is also responsible for providing Ethos with random data using a Random RPC.

We store Ethos's filesystem on Dom0 at /var/lib/ethos/*domainName*. Thus this directory on Dom0 is equivalent to Ethos *domainName*'s /. We use this to our advantage; Ethos applications are presently cross-compiled using a Go compiler that runs on Linux. Installing a program is a matter of copying it to /var/lib/ethos/*domainName*/programs/. We also have all of the tools available to us in Linux when we need to initialize or repair the directories and files that make up an Ethos filesystem.

Mixins and libraries We have written much of the code in Ethos in the form of libraries. Many functions useful in the kernel are also useful in user space. For example, we have a string manipulation library that is linked both into the kernel and into our user-space C library. Our abstract data type library provides another example. Some of this code is used in the Ethos kernel, Ethos user space, and Linux user space. We wanted to maximize this code reuse, but found some functions to be problematic. For example, what if a function needs to print a value (perhaps to facilitate debugging)? This is a very different operation whether taking place in the Ethos kernel or user space.

To solve this dilemma, we write mixins (named after the term's use in object-oriented programming). Mixins are small functions that perform kernel- or user-space-specific work. We have written a mixin library for the Ethos kernel, Ethos user space, and Linux user space. To solve the dilemma above, one must simply link against the appropriate mixin library. Each such library provides a different mixinPrint function, whose existence is assumed by any of our libraries that must print a value. Other mixin functions provided include mixinExit and mixinGetTimeOfDay.

Many of our libraries can be tested independently of their use in the Ethos kernel. For example, we extensively tested and profiled using gprof our allocator, libxalloc, long before we integrated it into the Ethos kernel. Furthermore, several of the libraries that we link into the Ethos kernel have unit tests we run independently of the kernel. Our RPC encoding library, libetn, has 34 unit tests and 42 benchmarks, each of which run outside of the kernel. We have found it beneficial to perform such tests outside of the kernel, because user space is a much more forgiving environment.

System testing So far, we have discussed testing libraries before we link them into the kernel. Here we discuss system testing. Running Ethos within Xen aids in testing the kernel. We have written a series of tests that exercise the system calls that Ethos provides. For each test, a script runs in Dom0 that sets up the environment for the test. Next, the script boots Ethos which executes a test application. After the test application makes one or more system calls it exits, and the Dom0 script checks its results. Thus each test is automated: we generally run all 73 of our tests in series to help detect regressions during development. We have found the ability to run these focused, automated tests to be another benefit of developing on top of Xen.

4. The impact of virtualization on assurance

Targeting a VMM is not without its pitfalls. The VMM itself can have bugs, and example code might also be flawed. We worked with the Xen team to fix one such problem in 64-bit MiniOS[3]. We also occasionally encounter performance regressions when updating our Xen installation to the most recent packages provided by the Fedora project. The cause of these were sometimes difficult to identify, although we found the Fedora and Xen communities helpful.

One of the reasons we chose to target the Xen VMM over other virtualization platforms is because Xen is a bare-metal hypervisor. This is beneficial from an assurance point of view as it removes a full intermediary OS from between the VMM and hardware. However, assurance concerns do arise in the current implementation of Xen.

Xen's privileged domain, Dom0, has direct access to (1) hardware I/O devices and (2) the memory of other Virtual Machines (VMs) [19]. To reduce vulnerabilities due to (1), Ethos encrypts data sent to communication devices and file systems [7]; similar techniques could secure keyboards and displays [32]. Vulnerabilities from (2) arise from DMA accesses (which use physical addresses) and because Dom0 has the ability to migrate VMs. In the former case, an I/O Memory Management Unit (IOMMU) can preserve confidentiality and integrity. In the latter case, Xen could encrypt VM pages prior to Dom0 access. Thus these security risks could be largely mitigated. An exception is availability, but unlike failures in confidentiality and integrity, availability failures are always apparent.

The size of Xen's code base and its exposure to Dom0 currently are impediments to building a highly secure implementation of Ethos. Orthogonal work to reduce vulnerabilities in Xen, disaggregate Dom0 [9, 31], produce alternative hypervisor technologies (e.g., Nova [42]), and verify hypervisors (e.g., MinVisor [10]) will help us address the limitations of our current research prototype.

If Ethos's design eventually proves to have sufficient strength and usability, it will be desirable to reimplement the Ethos research prototype in a way which can result in high assurance. This includes a microkernel implementation [1, 12, 30, 43, 44, 47], a minimalist VMM, and proof of correctness [22]. As we begin this work, we may also find it time to reimplement some of the techniques described here in a more traditional way. Given that we are still working on user-space evaluation, such a reimplementation would be premature—our strategy has allowed us to focus on our research interests.

5. Security through layering

At first glance, our notion of laziness does not appear to be a virtue for application programmers, but we hope to show with Ethos that this does not have to be the case. It is too difficult to write robust[4] programs on modern operating systems. Evidence indicates that the reason for this is that too much is left to application programmers and administrators. For example, Georgiev et al. discussed a Chase mobile banking application that did not validate certificates properly [17]. This made the application vulnerable to Man-in-the-Middle (MitM) attacks, a devastating vulnerability.

Organizations such as Chase face a trade-off: they can bring software to market quickly and maintain competitiveness, or they can be more thorough, possibly loosing customers who expect quick innovation. Perhaps the latter is more advantageous in the long run, but customers seem to expect the former. Of course, in hindsight one might tell Chase "*make sure you test certificate validation*". However, the reality is that there are just too many security-critical details to test in each application.

Of course, we have seen this pattern before. Buffer overflows were found difficult to test for in every application, and applications benefit in aggregate when buffer overflow protections are moved to their programming language. Once this is done, the protection must only be checked once—in the language. This idea has been applied to OSs before too: a similar philosophy drove the move from cooperative to preemptive multitasking. (There still are particular systems which benefit from deterministic cooperative multitasking and particular programs that require the control of lower-level languages; likewise, we expect Ethos to be sub-optimal for some non-security-critical applications.) Others have investigated raising the abstraction level of OSs: Ford decreased the application code necessary to maintain separate control and data connections with Structured Stream Tranport (SST) [14], and Bittau et al. moved toward ubiquitous transport-level encryption with tcpcrypt [5].

Moving protections into the OS shrinks the Trusted Computing Base (TCB). seL4 has been formally verified, but an application running on seL4 that relies on a Transport Layer Security (TLS) library would need to have that library verified too. The same is true of any library that provides encryption, authentication, or other security services. Systems that allow applications to build security in an ad hoc manner run the risk of becoming exceedingly difficult to assure. A model that provides strong security services once—at the OS level—maintains more simplicity. This is the strategy we have adopted for Ethos. (We are not claiming that Ethos is antagonistic to seL4. In fact, once we are satisfied with Ethos' interfaces, it might be beneficial to port them to seL4.)

Ethos guarantees many security properties—in general these fall into two classes, *security services* and *security hole avoidance*. Security services include authentication, authorization, and encryption. Security hole avoidance eliminates dangerous programming

[3] http://lists.xen.org/archives/html/xen-devel/2012-10/msg01792.html

[4] We define *robust* as a characteristic where a program continues to operate as intended even in the face of an intelligent adversary.

Vulnerability	Protection
Missing authentication for critical function	P3, P6
Missing encryption of sensitive data	P7
Use of a Broken or Risky Cryptographic Algorithm	P2, P3, P4, P7
Use of hard-coded credentials	P4
Execution with unnecessary privileges	P1†
Allow excessive authentication attempts	P4
Use of a one-way hash without a salt	P4
Incorrect permission assignment for critical resource‡	P2, P5

† in conjunction with Ethos' authorization system
‡ with respect to authentication secrets

Table 2: How Ethos protections address common vulnerabilities

pitfalls in such areas as race conditions, aliases, and input. We approach these two security property classes differently. Security services are implicitly provided by Ethos, so that they are ever-present, and thus do not depend on application programmers. Security holes are removed-by-design, by analyzing the security holes which exist on other systems and creating alternative semantics which do not have these security-hole-inducing pitfalls. Of course, not all security properties can be solved by careful layering, but the opportunity here is very large.

Ethos' security services are compulsory, meaning that programmers cannot incorrectly invoke or bypass them. Ethos protections include:

P1 Processes cannot change owners; instead, processes spawn special children that run as a different owner from inception

P2 Applications do not have access to secret keys; instead, Ethos isolates keys and provides access to cryptographic operations through system calls

P3 All network connections are authenticated

P4 Authentication uses strong techniques

P5 Confidentiality of authentication databases is not essential to security because Ethos uses public-key cryptography

P6 All communication made (client-side/local user) or received (server-side/remote user) are subject to authorization based on the requesting host and user

P7 All data written to disk or network devices is protected using strong cryptography

Protections P1–P7 defeat several classes of bugs that result in security holes. Of the CWE/SANS' *Top 25 Most Dangerous Software Errors* [27], Ethos' protections address eight error classes, or 32%. Table 2 shows the correspondence between Top 25 errors and Ethos' protections.

An example of an Ethos security service is its signature service [33], which supports P2. Ethos maintains secret signature keys and does not release them to applications. This model of isolation makes it impossible for an application to divulge a key, whether due to user error, programmer error, or compromised applications [11, 21, 26]. In order perform a digital signature, an application must invoke Ethos' sign system call. Beyond key isolation, this allows Ethos to apply its authorization controls to the production of signatures. This is useful to prevent a weakly-trusted application—such as a game found on the Internet—from signing something like a bank transfer request.

Parsing is vulnerable to very subtle attacks, and provides an example of security hole avoidance in Ethos. Dalí attacks exploit a single file which conforms to two different formats, and thus two different meanings [6]. Chameleon (similar to Dalí) and Werewolf attacks target file type inference and parsing, respectively. These attacks evade antivirus protections—which must often determine the type of a file before analyzing it [20]—due to type ambiguity. Cross-site Scripting (XSS) filtering is likewise impeded by this ambiguity [4]. Ethos addresses these problems with a type system. All reads and write in Ethos—whether to a file, Inter-Process Communication (IPC) stream, or network connection—are restricted to be well-formed with respect to a declared type. Ensuring programs receive only well-formed input reduces the LoC required to handle invalid input.

6. Related work

Microkernels reduce an OS kernel to the bare minimum necessary, and relegate many traditional kernel functions to isolated user-space processes [1, 12, 16, 22, 44]. This can ease the development of new OSs, as the microkernel itself does not need to be rewritten. Early critiques on the performance of microkernels have largely been addressed [24] and one microkernel, seL4, has been formally verified [23]. We view microkernels as complementary to the techniques we describe here.

Extensible kernels allow applications to directly manage resources. Examples include the Exokernel [12] and SPIN [3]. Such application-specific management may improve system performance. Like microkernels in general, extensible kernels are complimentary to our techniques.

The Flux OS Toolkit (OSKit) [15] provides a reusable, low-level OS infrastructure that can serve as the basis of new research OS kernels. The idea behind OSKit is that existing modular components can be combined with novel components to produce a new OS. OSKit includes modules for kernel bootstrap loading, memory management, filesystems, a minimal C library, and device drivers. Some OSs save work by providing a compatibility layer, allowing them to use drivers designed for another OS. Haiku adopted this strategy and can make use of some FreeBSD network drivers.

Several OSs have targeted virtual machines. Qubes [39] is an OS architecture that leverages Xen's properties of isolation in an attempt to better segregate security-critical applications from routine applications. Mirage investigated compiling applications into OS kernels which run directly on top of Xen [25]. Mirage pursues a different approach than Ethos: Ethos simplifies programming by providing high-level, mandatory interfaces, whereas Mirage provides more control.

In some ways, Ethos' higher abstractions resemble middleware such as OSGi [18]. Middleware often contains bugs that give rise to security holes [17]. Of course, OSs have bugs too, but moving abstractions into the OS allows for a simpler overall design. Furthermore, when the OS subsumes an interface, the interface benefits from an OS' property of complete mediation [40]. Other OSs share Ethos' strategy of higher abstractions; for example, Taos and Singularity identify to applications the remote principal associated with a network request [48, 49]. Indeed, virtualization allows us to reconsider the "end" in the end-to-end principle [41]. Instead of considering the terminating point an application, we consider it one of many OSs running on a computer, with each OS providing an environment appropriate for its applications.

7. Conclusion and further work

It has been estimated that the cost of developing a general-purpose operating system is in the billions of dollars [28]. We estimate that we saved writing approximately 30,000 lines of kernel code through the techniques described here. Many more LoC were saved by specializing the OS to our particular requirements. Ethos in-

cludes only strong protections; it does not spend code supporting weak authentication mechanisms, for example. Finally, we avoided the need to write a large number of device drivers. Thus we propose that virtualization gives rise to the more expedient development of *specialized* operating systems, such as Ethos. In addition, virtualization has provided us with kernel debugging and profiling, two features we would likely not yet have implemented had we not written Ethos on top of Xen. We have found other benefits as well, for example it is quicker to restart a kernel that crashes inside of a virtual machine than it is to wait for real hardware to initialize.

The freedom to easily experiment with Ethos' system call interface has benefited applications. In one study, we wrote a messaging application and compared it to Postfix. Whereas Postfix contains roughly 2,000 LoC to support robust networking, our application dedicates zero lines of code to network confidentiality, integrity, authentication, and authorization. Instead, it benefits from OS-level protections, as all Ethos applications do.

One interesting line of research would be to reimagine OSKit as a Xen-based augmentation to MiniOS. The obvious disadvantage of our approach to using MiniOS is that our modifications to the kernel make it difficult to merge future MiniOS work back into Ethos. Breaking MiniOS into a series of OSKit-like components would encourage additional research into custom kernels, and would allow those kernels to more easily incorporate new developments in MiniOS. Another area we are interested in is building an Xen-based kernel using a higher-level systems language such as Go.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Proceedings Summer USENIX*, 1986.

[2] 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*, 2003.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[4] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA*, 2008.

[5] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proceedings of the 19th USENIX conference on Security*, 2010.

[6] F. Buccafurri, G. Caminiti, and G. Lax. Fortifying the Dalí attack on digital signature. In *Proceedings of the 2nd International Conference on Security of Information and Networks*, 2009.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[9] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *SOSP*, 2011.

[10] M. Dahlin, R. Johnson, R. Krug, M. McCoyd, S. Ray, and B. Young. Toward the verification of a simple hypervisor. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, 2011.

[11] P. Dasgupta, K. Chatha, and S. K. S. Gupta. Viral attacks on the DoD common access card (CAC). available at `http://cactus.eas.asu.edu/partha/Papers-PDF/2007/milcom.pdf`, 2009.

[12] D. R. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, 1995.

[13] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *CCS*, 2012.

[14] B. Ford. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007.

[15] B. Ford, K. V. Maren, J. Lepreau, S. Clawson, B. Robinson, and J. Turner. The flux OS toolkit: Reusable components for OS implementation. In *HotOS*, 1997.

[16] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *ACM SIGOPS European Workshop*, 2000.

[17] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS*, 2012.

[18] R. Hall and H. Cervantes. An OSGi implementation and experience report. In *Consumer Communications and Networking Conference*, pages 394–399, jan. 2004. doi: 10.1109/CCNC.2004.1286894.

[19] J. Harper. Re: Questions about attacks on Xen. xen-devel mailing list, 2012.

[20] S. Jana and V. Shmatikov. Abusing file processing in malware detectors for fun and profit. In *Proc. IEEE Symp. Security and Privacy*, 2012.

[21] G. Kent and B. Shrestha. Unsecured SSH—the challenge of managing SSH keys and associations. White paper, SecureIT, 2010.

[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[23] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 2010.

[24] J. Liedtke. Improving IPC by kernel design. In *SOSP*, 1993.

[25] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863114`.

[26] J. Marchesini, S. W. Smith, and M. Zhao. Keyjacking: the surprising insecurity of client-side SSL. Technical report, Dartmouth College, 2004.

[27] B. Martin, M. Brown, A. Paller, and D. Kirby. 2011 CWE/SANS top 25 most dangerous software errors. Technical report.

[28] A. McPherson, B. Proffitt, and R. Hale-Evans. Estimating the total development cost of a linux distribution. Technical report, 2008.

[29] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference On Virtual Execution Environments*, 2005.

[30] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba—A distributed operating system for the 1990s. *Computer*, 1990.

[31] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *VEE*, 2008.

[32] R. Perez, L. van Doorn, and R. Sailer. Virtualization and hardware-based security. *Proc. IEEE Symp. Security and Privacy*, 2008.

[33] W. M. Petullo and J. A. Solworth. Digital identity security architecture in Ethos. In *Proceedings of the 7th ACM workshop on Digital identity management*, 2011.

[34] R. Pike. System software research is irrelevant, 2000.

[35] M. Prandini, M. Ramilli, W. Cerroni, and F. Callegati. Splitting the HTTPS stream to attack secure web connections. *IEEE Security and Privacy*, 2010.

[36] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *CACM*, 1974.

[37] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *HotOS*, 2007.

[38] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *PLDI*, 2009.

[39] J. Rutkowska and R. Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 2010.

[40] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[41] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 1984.

[42] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys*, 2010.

[43] A. S. Tanenbaum and M. F. Kaashoek. The Amoeba microkernel. In *Distributed Open Systems*. 1994.

[44] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation 3/e*. 2006.

[45] N. Vratonjic, J. Freudiger, V. Bindschaedler, and J.-P. Hubaux. The inconvenient truth about web certificates. In *The Workshop on Economics of Information Security (WEIS)*, 2011.

[46] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. 2000.

[47] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[48] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *SOSP*, 1993.

[49] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. R. Simon. Authorizing applications in Singularity. In *EuroSys*, 2007.