# Towards Application Security on Untrusted Operating Systems

Dan R. K. Ports
*MIT CSAIL & VMware, Inc.**
drkp@csail.mit.edu

Tal Garfinkel
*VMware, Inc.*
talg@vmware.com

## Abstract

*Complexity in commodity operating systems makes compromises inevitable. Consequently, a great deal of work has examined how to protect security-critical portions of applications from the OS through mechanisms such as microkernels, virtual machine monitors, and new processor architectures. Unfortunately, most work has focused on CPU and memory isolation and neglected OS semantics. Thus, while much is known about how to prevent OS and application processes from modifying each other, far less is understood about how different OS components can undermine application security if they turn malicious.*

*We consider this problem in the context of our work on Overshadow, a virtual-machine-based system for retrofitting protection in commodity operating systems. We explore how malicious behavior in each major OS subsystem can undermine application security, and present potential mitigations. While our discussion is presented in terms of Overshadow and Linux, many of the problems and solutions are applicable to other systems where trusted applications rely on untrusted, potentially malicious OS components.*

## 1 Introduction

Commodity operating systems are tasked with storing and processing our most sensitive information, from managing financial and medical records to carrying out online purchases. The generality and rich functionality these systems provide leads to complexity in their implementation, and hence their assurance often falls short.

Many solutions have been proposed for enhancing the assurance of these systems. Some use microkernels to contain potentially malicious behavior by placing major OS and application components into separate isolated processes [11]. Others, such as NGSCB (formerly Palladium) [3], Proxos [12], XOM [7], and Overshadow [2], attempt to retrofit orthogonal, higher assurance execution environments alongside a commodity OS, allowing part or all of an application to run in a protected environment, but still use OS services. Unfortunately, while these systems provide CPU and memory isolation in the face of OS compromise, the implications of continuing to rely on OS services if they turn malicious are poorly understood.

We explore this problem and potential solutions in the context of Overshadow, a virtualization-based system we developed that protects applications in a VM from the guest operating system in that VM. Overshadow attempts to maintain the secrecy and integrity of an application's data even if the OS is completely compromised. For each major OS component, we examine how malicious behavior could undermine application secrecy and integrity, and suggest potential mitigations. While we present our analysis and solutions in the context of Linux and Overshadow, they are more generally applicable to any system attempting to secure application execution in the face of a compromised OS.

We begin with a review of systems that enforce isolation between protected applications and untrusted operating systems in Section 2. Next, we explain how a malicious OS can subvert them, using false system call return values to trick an application into revealing its secrets, and argue for a solution based on a verifiable system call interface in Section 3. Finally, we examine the implications of making specific OS components untrusted, and propose defenses against possible attacks, in Section 4.

## 2 Isolation Architectures

Traditional commodity operating systems are monolithic programs that directly manage all hardware, so any OS exploit results in total system compromise. A variety of architectures have been proposed to mitigate the impact of an OS compromise by providing memory and CPU isolation using a separate layer below the commodity OS, either in the form of a virtual machine monitor, microkernel or architectural extensions — taking sole responsibility for protection out of the hands of the OS.

These architectures prevent a compromised OS from reading or modifying application CPU and memory state. We are primarily interested not in these attacks but rather what comes next, *i.e.* once we have prevented direct attacks on application state, how we can prevent higher-level attacks based on modifying the semantics of system calls. However, CPU and memory isolation is a prerequisite for our work, so we begin with a survey of these isolation architectures.

*Work done during an internship at VMware, Inc.

Microkernel-based architectures, such as those based on L4 [5], divide both the application and OS into multiple components, placing each in its own process. In Proxos, a virtual-machine-based system, trusted components including the protected applications and a trusted "private OS" are placed in a single VM, that runs alongside another VM running a commodity OS [12]. A proxy forwards the application's system calls to either the private OS or commodity OS, depending on an application-specific security policy. NGSCB takes a similar, but less backwards-compatible approach, by requiring the application to be refactored into a trusted part which runs in its own secure compartment and an untrusted part which runs on the commodity OS [3]. XOM is a processor architecture that isolates protected applications from each other and the operating system using a combination of register-level tagging and cryptographic techniques [7].

While seemingly different, all these architectures provide the same basic property: some or all of the application is run in a separate protection domain from the OS. This protects the secrecy and integrity of resources like registers and memory, and ensures control flow integrity, *i.e.* control flow cannot be altered except via well-defined protocols such as signal delivery and system calls.

**Overshadow.** Overshadow [2] is a system we developed for protecting applications running inside a VM from the operating system in that VM. Overshadow works by leveraging the virtual machine monitor's control over memory mappings to present different views of memory pages depending on the execution context. When an application accesses its own memory, access proceeds as normal; if the OS or another application accesses it, the VMM transparently encrypts the memory page and saves a secure hash for integrity verification. This allows all resource management functionality to remain in the hands of the OS, without compromising secrecy and integrity: for example, the OS can swap memory pages to disk and restore them, but cannot read or modify their contents.

To adapt applications to this new execution environment, without requiring any modifications to the application or OS, we add a *shim* to each application at load time. The shim is user-level code (akin to a shared library), but communicates directly with the VMM via a *hypercall* interface. The shim and VMM together provide secure control transfer by saving execution state in protected memory when an application is interrupted, and restoring it when the application resumes execution. The VMM also redirects system calls to the shim, which adapts their semantics as necessary for compatibility; for example, data being passed to the OS is automatically copied to unprotected memory so that the OS can access it. In the extensions to Overshadow we propose in Section 4, we make use of this system call interposition to help implement system calls securely.

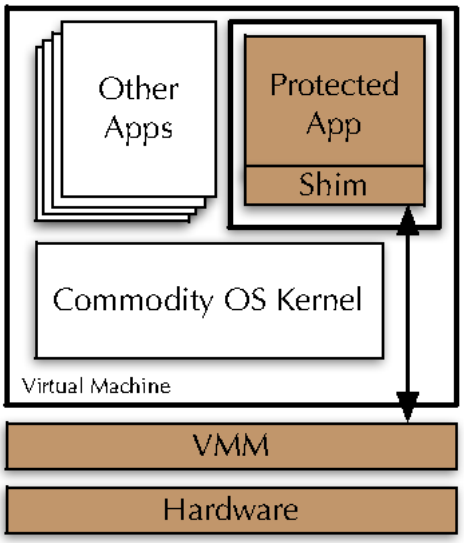


Figure 1: Overshadow architecture. Trusted components are indicated with a shaded background. In addition to the standard virtualization barrier that isolates entire VMs from each other and the monitor, Overshadow adds a second virtualization barrier that isolates protected applications from other components inside the same VM.

Figure 1 shows the components required to protect an application with Overshadow. In addition to the application itself, the shim, VMM, and hardware are trusted. The OS kernel and all other applications, including privileged daemons and other system services, are untrusted.

Overshadow uses a simple protection model based on whole-application protection. Each application runs in its own secure compartment identified by a unique *security ID* (SID). Each process or thread thus has an associated SID. Every resource, such as a memory region or file, has an owner SID. Access to a resource is permitted only if its SID is the same as that of the currently-executing thread. Though simple, this model is sufficient for many interesting applications. It would be relatively straightforward to extend this model to support trust relationships between applications and more sophisticated access control for shared resources.

## 3 Implications of a Malicious OS

The protection architectures in the previous section guarantee that applications that do not interact explicitly with the OS execute correctly, because the integrity of code and data is protected, and control flow proceeds as normal. Assuming the basic protection architecture is sound, no loss of integrity or data leakage is possible, although availability is not guaranteed because the OS could simply stop scheduling the application.

However, any non-trivial program makes system calls, and this presents an opportunity for a malicious OS to influence the program's data and control flow by manipulating system call results. To take a simple example, if a multithreaded program relies on the OS to implement a mutual-exclusion lock, the OS could grant the lock to two threads at once. Since the OS also handles scheduling, this has the potential to create a race condition even in correctly written code, with arbitrarily bad consequences.

Characterizing the security properties that are possible when the operating system is malicious is challenging. Ultimately, we would like to provide a high-level guarantee like "sensitive data is never exposed to unauthorized parties." However, we would like to operate with unmodified applications, treating them as black boxes, and therefore cannot make such statements about application semantics. Indeed, a buggy application might expose its own sensitive data (*e.g.* via a buffer overflow), even if the OS behaves correctly. Thus, our problem must be one of *ensuring that applications continue to run normally* (or fail-stop) even if the OS behaves maliciously, rather than one of protecting application data.

## 3.1 Approaches to Ensuring Security

For programs to run normally, we must guarantee that the action performed and value returned by each system call conforms to the application's model of how system calls behave. Ideally, this model would be the same as the standard OS contract for system calls (*e.g.* the POSIX specification). However, modeling all behaviors of every system call would be tantamount to reimplementing nearly the entire OS. Moreover, there may be behavior that technically complies with the specification, but still violates application assumptions.

Instead, we propose to develop a new specification consisting only of *safety properties*, *i.e.* a model of normal OS semantics that pertains only to security, not to availability. By weakening the specification to only provide safety properties, it becomes easier to hold the OS to its contract. For example, with the mutex system call described above, we might guarantee only that if a lock is granted, it is held by no other thread, saying nothing about availability or fairness. Providing weaker semantics does mean that it is impossible to guarantee correctness of arbitrary unmodified applications, but we anticipate that the additional requirements will not pose a large burden for developers, and correspond in many cases to good programming practice on a correctly-functioning OS.

There are three fundamental methods for ensuring that system calls are executed correctly:

- We can **disallow** use of the system call in security-critical code, permitting applications to use it "only at their own risk." Clearly, this is to be avoided from a compatibility perspective. However, certain system calls are so intimately related to the OS implementation that their correctness cannot be guaranteed — for example, those related to kernel modules or scheduling policies.
- We can **emulate** the system call in trusted code. This option should also be used sparingly, since reimplementing system calls adds to the size of the TCB. Nevertheless, it may be the best option for simpler system calls, such as those related to time.
- We can **verify** the results of the system call after the OS has executed it. This is our preferred approach, since verifying a system call's safety properties can often be substantially simpler than emulating it. This is the approach that Overshadow and XOM already use for protecting memory: they delegate memory management to the OS, but verify hashes at access time to ensure that the correct data is in the correct memory location.

Returning to the mutex system call example, we can verify that only one thread holds the lock at once using a flag stored in a protected shared memory region to indicate whether the lock is held. Each thread updates the flag as it acquires or releases the lock, and verifies that the flag is not already set when it acquires the lock. This is a simple procedure that ensures the key safety property of locking while still delegating the rest of the implementation details (waking up the right process when the lock is available, ensuring fairness, etc.) to the OS.

We propose to implement emulation and verification using Overshadow's existing system call interposition mechanism, which redirects control to the shim, a trusted component, whenever an application makes a system call. However, these techniques can be used independently of Overshadow, by using a different interposition technique, or by modifying the application.

# 4 Attacks and Mitigations

We now turn our attention to specific OS features that applications commonly depend upon. For each feature, we examine how a malicious OS might use it to mount an attack on an application, and whether that functionality can be securely delegated to the OS using a verifiable interface. For concreteness, we use examples based on a Linux OS, but although the details may differ, the components we discuss are common to most operating systems.

## 4.1 File System

One of the most important services provided by the OS is access to persistent storage; it is also particularly critical for security, since both the program code and its (potentially sensitive) data are stored on the file system.

### 4.1.1 File Contents

**Potential attacks.** Protection is clearly needed for file contents. If files are stored unprotected, a malicious operating system could directly read an application's secret data as soon as it is written to disk. A malicious OS could also tamper with application binaries, replacing an application with code that simply prints out its sensitive data. It might also launch a replay attack, reverting a file to an earlier version, perhaps replacing a patched application with an earlier version that contains a buffer overflow.

**Proposed solution.** Most protection architectures already provide some protection for file contents, thereby thwarting these attacks. For example, Overshadow's cryptographic secrecy and integrity protection extends to files stored on disk as well as memory. This is accomplished by translating all file I/O system calls into operations on memory-mapped file buffers. Since these buffers consist of memory pages that are shared between the application and the kernel, they are automatically encrypted and hashed when the OS flushes them to disk. Like memory regions, all files are encrypted with the same key, known only to the VMM and stored securely outside the VM; access control is independent of key management.

To defend against tampering, reordering, and replay attacks on file contents, Overshadow maintains *protection metadata* for each file, consisting of a secure hash of each page in the file, in addition to the randomly-chosen block cipher initialization vectors. This protection metadata is protected by a MAC and freshness counter, and stored in the untrusted guest file system.

### 4.1.2 File Metadata

**Potential attacks.** More subtly, file *metadata* needs to be protected, including file names, sizes, and modification times. Many designs omit this aspect, relying on the OS for services such as pathname resolution. As a result, a malicious OS could perform a pathname lookup incorrectly. Even a system that protects file contents may be subverted if the OS redirects a request for a protected file to a *different* but still valid protected file. For example, with Overshadow's file protection mechanism described above, such an attack could succeed if the OS also redirected the access to the protection metadata file that contains the hashes to verify the file. It can only redirect file lookups to valid, existing protected files, but this still opens many possibilities for attack, such as redirecting a web server's request for `index.html` to its private key file instead.

**Proposed solution.** We propose using a trusted, protected daemon to maintain a secure namespace, mapping a file's pathname to the associated protection metadata. Applications can communicate with this daemon over a protected IPC channel (as described in Section 4.2), requesting directory lookups when files are opened, and updating the namespace when files or directories are created, removed, or renamed. Each file or directory can also have an associated list of SIDs (*i.e.* applications) that are allowed to access it. Maintaining this namespace requires adding code for directory lookups to the TCB, but this can be far smaller than a full file system implementation. This design was proposed in Overshadow [2], and also used in VPFS [14], a similar file system architecture for L4 that uses a small trusted server and an untrusted commodity file system to reduce TCB size.

Alternatively, as noted in [14], storing a hash of a file's pathname in its header provides a much simpler way to verify that pathnames are looked up correctly, but does not allow directory contents to be enumerated.

Similar ideas are used by other systems that build secure storage on an untrusted medium. VPFS also uses a trusted server to store file system metadata, although it uses different techniques to secure file contents [14]. SUNDR [6] and Sirius [4] are distributed file systems that use client-side cryptography to avoid trusting the file server; because they have no trusted storage, they cannot guarantee freshness, and are therefore subject to *fork attacks*, where the file server presents different versions to different clients. Like TDB [8], we have available a small amount of trusted storage (in our case, in the VMM) that can be used to guarantee freshness.

## 4.2 Inter-Process Communication

A trusted inter-process communication mechanism is a key component of a secure system. In addition to protecting application communications, it is a useful building block for constructing other secure components; for example, it is necessary for communicating with the file system namespace daemon.

**Potential attacks.** IPC channels provided by the OS are insecure, and thus face all of the standard problems inherent in communication over an untrusted channel. A malicious OS might spy on IPC messages between protected processes, or might tamper with, drop, delay, reorder, or spoof messages.

Many attacks are possible as a result. For example, a secure application might consist of a database of sensitive information such as credit card numbers that is accessible only through a restricted web interface. A malicious OS could observe the credit card numbers as they are transmitted over the web server's IPC connection to the database server, or it could tamper with the database by sending spoofed requests over the IPC connection.

More subtle attacks are also possible, much like the attacks on file metadata. Rather than directly inspect the contents a protected application's IPC channel, the OS might redirect the connection to point to a different process which would then expose the data, such as

`/bin/cat`. The OS could also simply refuse to deliver any messages between two processes.

**Proposed solution.** One way to provide secure IPC is to implement it entirely in the trusted layer, by setting up a message queue in the VMM. Processes could then enqueue messages or check for pending messages via secure hypercall. A problem with this approach is that it is impractical for applications to poll for messages, since this either requires waking up each process regularly, or tolerating a high message latency. However, we can use the guest operating system to provide asynchronous notifications: after sending a message through the VMM, the sender also sends the receiving process a signal through the guest OS. Because the guest OS does not handle message data, it cannot impact confidentiality, integrity, or ordering; the OS is relied upon only for availability.

However, although this approach is suitable for small, infrequent messages, it is not ideal for large data transfers, both because of the need to copy data into and out of the VMM, and to keep VMM complexity to a minimum. Instead, we can use shared memory regions for most of the communication, using VMM-assisted communication only for bootstrapping the secure channel. Specifically, a protected process wishing to communicate with another process in the same compartment would create a shared memory region (*e.g.* using `mmap`), and populate it with a pair of message queues. Using Overshadow's protection mechanism for memory-mapped file contents, the OS cannot read or modify the contents of the shared memory region. However, the OS manages the namespace of these shared memory regions, so it might still attempt to map in a different region, such as the one corresponding to a different IPC channel. To defend against this, the sender can place a random nonce in the memory region, and communicate it securely to the recipient through the VMM. As before, the untrusted OS's signals can be used as asynchronous notifications.

Implementing IPC in this way guarantees secrecy, integrity, and ordering, but there cannot be any guarantees that messages are received in a timely manner (or at all) when the operating system could delay or terminate one of the processes involved. We could have added acknowledgements to our message-passing protocol, blocking the sender until the receiver acknowledges the message, but chose not to because the OS could still stop the receiving process after it acknowledges the message but before acting on it. Instead, we require that applications not assume messages have been received unless they implement their own acknowledgement protocol. This is sound practice even with a correctly functioning OS, as the receiving process might be slow or have crashed.

## 4.3 Process Management

The OS is responsible for the management of processes, including starting new processes and terminating existing processes. In addition, it manages process identities, which applications rely on for directing signals and IPC messages. This opens several avenues of attack.

**Potential attacks.** Although the OS cannot interfere with program execution contexts and control flow during normal operation, it might be able to do so when a new process is started. For example, when a process forks, it might initialize the child's memory with malicious code instead of the parent's, or set the starting instruction pointer to a different location. Signal delivery also presents an opportunity for a malicious OS to interfere with program control flow, since the standard implementation involves the OS redirecting a program's execution to a signal handler.

A malicious OS might try to redirect signals, process return values, or other information to the wrong process. It might attempt to change a process's ID while it is running, or send the wrong child process ID to a parent.

**Proposed solution.** Solutions for securing control flow for newly-created processes are relatively well-understood. Overshadow interposes on `clone` and `fork` system calls to set up the new thread's initial state. This includes cloning the memory integrity hashes and thread context (including the instruction pointer), thereby ensuring that the new thread can only be started in the same state as its parent.

To ensure that signals are delivered to the correct entry point, Overshadow also maintains its own protected table of the application's signal handlers. It registers only a single signal handler with the kernel, which immediately makes a hypercall to the VMM. The VMM then securely transfers control to the appropriate signal handler.

We can address the problems related to the OS managing process identity by using an independent process identity in conjunction with the secure IPC mechanism of Section 4.2. Whenever a new process is created, it is assigned a secure process ID (SPID) to identify it for secure IPC purposes; this is an identifier that is conceptually independent of the OS's process ID, although with a correctly functioning OS there will be a one-to-one relationship. When a process is forked, its SPID is communicated to the parent, along with the OS's process ID, via a secure IPC message. When one process wants to send another a signal, it sends a secure IPC message identifying itself and the signal. Similarly, when a process exits, it sends its return value securely to its parent.

## 4.4 Time and Randomness

**Potential attacks.** The operating system maintains the system clock, which means that security-critical applica-

tions cannot rely on it. A malicious OS could speed up or slow down the clock, which could allow it to subvert expiration mechanisms in protocols like Kerberos [9] or time-based authentication schemes. It might also cause the clock to move backwards, an unexpected situation that could expose bugs in application code.

In addition, the standard system source of randomness comes from the OS, making it unsuitable for use in cryptographic applications. A malicious OS could use this to control private keys generated by an application, or defeat many cryptographic protocols.

**Proposed solution.** We see little solution other than to create a trusted clock and source of secure randomness. In our system, these would be implemented in the VMM, and time-related system calls and access to `/dev/random` would be transformed into hypercalls.

Although this requires adding additional trusted components to the system, the TCB expansion is not significant because the VMM or microkernel likely already includes time and entropy services for its own use. However, keeping time perfectly synchronized between the guest OS's clock and the VMM's can be challenging even with a correctly functioning OS [13].

## 4.5 I/O and Trusted Paths

**Potential attacks.** An application's input and output paths to the external world go through the operating system, including display output and user input. The OS can observe traffic across these channels, capturing sensitive data as it is displayed on the screen, or input as the user types it in (*e.g.* passwords). It could also send fake user input to a protected application, or display malicious output, such as a fake password entry window.

Network I/O also depends on the operating system, but this poses less of a problem because many applications already treat the network as an untrusted entity. Cryptographic protocols such as SSL are sufficient to solve this problem, and are already in common use.

**Proposed solution.** There are many complex issues inherent in designing a secure GUI, such as labeling windows and securing passphrase entry; many of these have been studied extensively in the context of multi-level secure operating systems [1, 10]. We do not address them here, but focus on the question of how to achieve a trusted path that does not rely on the operating system.

A simple approach that maintains backwards-compatibility with existing applications is to run a dedicated, trusted X server in the application's compartment. Overshadow's memory protection can ensure that only the application and the virtual graphics card can access the server's framebuffer in unencrypted form. This approach requires adding the entire X server and its dependencies to the application's TCB. The situation may not be as grim as the number of lines of code would suggest, however, because the attack surfaces are limited. The trusted display server's interfaces can be limited to a secure socket to the protected application and the virtual I/O devices, so an attacker cannot easily interact with it.

Nevertheless, we might still wish to avoid trusting the entire display server, and instead use an untrusted display server that manages the display without having access to the contents of windows. It seems possible to achieve this using a window system architecture where applications render their window contents into buffers, and the window server simply composites them. It is not clear, however, how to implement this in a way that maintains compatibility with existing applications.

## 4.6 Identity Management

The OS is responsible for managing a number of types of identities; we have already discussed the need to secure file system names and process IDs. Several others also exist, including user and group IDs and network endpoints (IP addresses, DNS names, and port numbers).

**Potential attacks.** These OS-managed identities are frequently used in authentication: applications often use the user ID of a local process or the IP address of a remote host to determine whether to grant access to a client. A malicious OS could cause a connection from an attacker to appear to be coming from a trusted local user or host.

**Proposed solution.** Applications should not rely on these identities for authentication or other security-critical purposes. Secure authentication can be implemented cryptographically for either local or remote connections. It may also sometimes be possible to securely authenticate a local connection simply by verifying that both endpoints are in the same secure compartment.

## 4.7 Error Handling

When a system call fails, the operating system returns an error code that, in addition to indicating failure, gives a reason for the failure. A malicious OS might return an incorrect error code, affecting a protected application's control flow. There are several types of violations. It is relatively straightforward to detect values that are clearly invalid according to the system call specification, such as a "bad file descriptor" error on a `fork` call. However, the OS might return a legitimate error code for an error that did not take place. Certain error codes can be verified because they correspond to functions that are implemented in or can be verified by trusted components. For example, if `open` returns a "no such file" error, the trusted namespace daemon (Section 4.1.2) can distinguish between cases where the file legitimately never existed and those where the OS is trying to conceal the file. Other error codes cannot easily be verified, such as returning

a "network unreachable" error on `connect`, even if no network error took place. In these cases, applications cannot rely on the error codes being accurate for safety.

Given that error checking is essential for constructing robust software, it may seem alarming that error codes cannot always be guaranteed accurate. However, applications frequently use error values in ways that do not affect the secrecy of sensitive data. In our experience, the most common responses to a failed system call are to ignore the error, retry the system call, or fail-stop — in most cases, applications are mainly concerned with whether the system call succeeded rather than the reason for failure. There are a few common exceptions, such as the "no such file" error mentioned above, but these errors are often verifiable and thus can be relied upon. Indeed, many of the errors that cannot be verified correspond to availability problems (*e.g.* "out of memory" or other resource limit errors), or untrusted components such as the network.

## 5 Conclusions

Our experience in building the Overshadow system for enhancing assurance of applications built on commodity operating systems has suggested two key lessons. First, systems looking to enhance OS assurance typically focus on core isolation mechanisms such as memory and CPU protection, but this is not sufficient to build a secure system. We observe that additional attention must be paid to how applications are affected by malicious OS behavior. To talk about "trusted" and "untrusted" parts of the OS interface without looking more deeply into what can and cannot be safely relied upon from the OS, is to casually dismiss a non-trivial problem.

Next, to address this problem, we observe that it is often easier to verify correct behavior than to implement that behavior. Many OS components can be refactored into an untrusted part that manages the system resource in question and a smaller trusted part that verifies the safety properties. In our experience, the trusted part can be made considerably smaller than the untrusted part: our implementation of Overshadow required under 20,000 lines of code to provide protection for application memory, files, and control flow, in contrast to the millions of lines of code that make up a typical OS [2]. We propose extensions to Overshadow that allow it to tolerate more complex malicious behavior from the OS, based on this principle of verifying system call results. Similar techniques can be used in other systems that rely on untrusted OS or application components.

## Acknowledgements

We are grateful to Mike Chen, E. Lewis, Pratap Subrahmanyam, and Carl Waldspurger for much hard work in designing and building Overshadow, and for many helpful discussions that led to this paper. Barbara Liskov provided valuable feedback on a draft of this paper.

## References

[1] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *Transactions on Software Engineering*, 16(2):608–618, June 1990.

[2] 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 *Proc. ASPLOS '08*, Seattle, WA, Mar. 2008.

[3] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Spectrum*, 36(7):55–62, July 2003.

[4] E.-J. Goh, H. Shacam, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. NDSS '03*, San Diego, CA, Feb. 2003.

[5] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. In *Proc. CollaborateCom '05*, San Jose, CA, Dec. 2005.

[6] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI '04*, San Francisco, CA, Dec. 2004.

[7] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. ASPLOS '00*, Cambridge, MA, Nov. 2000.

[8] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. OSDI '00*, San Diego, CA, Oct. 2000.

[9] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, Sept. 1994.

[10] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proc. USENIX Security '04*, San Diego, CA, Aug. 2004.

[11] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proc. EuroSys '06*, 2006.

[12] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. OSDI '06*, Seattle, WA, Nov. 2006.

[13] Timekeeping in VMware virtual machines. VMware, Inc. Technical White Paper, Aug. 2005. Available from `http://www.vmware.com/vmtn/resources/238`.

[14] C. Weinhold and H. Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *Proc. EuroSys '08*, Glasgow, Scotland, Apr. 2008.