Secure User I/O for Applications
on an
Untrusted Operating System

Joshua Harry Berlin
May 2, 2014

Department of Computer Science
The University of Texas at Austin

Committee: Emmett Witchel (Supervisor), Vitaly Shmatikov, and William H. Press
Acknowledgements

I would first like to thank my advisor Dr. Emmett Witchel. He dedicated significant time and effort to helping me learn and grow as a student and person. I would also like to thank the other students in OSA, especially Alan Dunn, for providing guidance and assistance in the face of sometimes daunting tasks.

Further I would like to thank my parents and family, without whom I would certainly not be where or who I am today.
1 Abstract

In this paper I discuss the topic of operating system trust, and specifically why it is important to remove the OS from the trusted computing base. I explore past work on running trusted applications on top of an untrusted OS including InkTag, and analyze why such systems need a mechanism for Secure User I/O. I introduce a mechanism for Secure User I/O that builds on a newer version of InkTag, taking advantage of the unique features of our system to provide an efficient and lightweight path for user input and output. This improved path for user I/O supports both graphical (X-windows) and terminal based applications with no modifications to application code. To our knowledge, this system is the first to provide a comprehensive solution for secure user I/O on an untrusted operating system without modification by the application developer.

2 Introduction

Computer systems play an increasing role in modern society. Many of the systems and applications we interact with on an everyday basis run on top of some form of commodity, off the shelf operating system. These operating systems are very complex: recent estimates place Windows at more than 50 million lines of code and the Linux kernel now exceeds 15 million lines of code [7, 12]. With codebases of this size, its almost guaranteed that bugs and security flaws will exist, and operating system manufacturers regularly release patches to fix bugs and exploits as they are discovered.

As a result of such inevitable vulnerabilities, no matter how well applications are designed, a single flaw or compromise in an operating system can result in the compromise of each application running on top of it. The amount of money and effort spent on the prevention and redress of malware, rootkits and viruses across popular computing platforms today is evidence of the significance of such vulnerabilities.

Existing research efforts to address operating system vulnerabilities rely on reducing the trusted computing base, the portion of the overall system that is integral to system integrity
and security. Although it might be conceivable to design and implement a minimal, secure operating system, doing so introduces many other problems in practice, especially since most of the applications that are used today are written for and take advantage of features from existing platforms. This work builds on top of InkTag[1], a system that enables users to run trusted applications inside an untrusted operating system. InkTag relies on the property that, although modern operating systems are complex behemoths, at the core of each is a set of basic, easily verifiable functionality such as managing an applications address space. InkTag is built on a virtualization architecture where an untrusted operating system runs on top of a trusted hypervisor, allowing the hypervisor to verify and enforce the correct behavior of the OS. One of the major contributions of InkTag is paraverification, a technique in which the untrusted OS, user applications and hypervisor work in tandem to verify the behavior of the OS. Based on the information provided to it by applications and the untrusted operating system, the hypervisor is able to detect anomalous operating system behavior and kill an uncooperative OS. The hypervisor verifies the behavior of an operating system but cannot provide any availability guarantees.

Although InkTag provides an effective system for running trusted applications in isolation, it lacks a mechanism for secure user I/O. Normal paths for user I/O are insecure in the context of an untrusted OS because the data involved passes in plaintext through the OS. Working on an untrusted operating system, users might worry about an OS eavesdropping on or modifying such communications. In this paper, I specifically focus on the design and evaluation of a system for providing secure user I/O.

I start by discussing the motivation for this work, including a more in depth discussion of the threat model this system attempts to address. In the next section, I present a technical background on the design of InkTag, including specifically why a separate mechanism is needed to provide secure user I/O. Subsequently, I give an overview of the design goals for the system and different scenarios we considered to achieve those goals. Following this, I discuss the implementation of the system and the methodology used for evaluating the performance of our design. Finally, I discuss related work.
3 Motivation

3.1 Overview

Computer systems are commonly used for casual tasks from browsing social networks to more important ones that involve sensitive and important user data such as interacting with banks and healthcare providers. Contemporary operating systems provide a set of primitives that can be useful for writing applications that accomplish these tasks; however, it’s very difficult as a user to know that an operating system isn’t acting maliciously behind the scenes. It’s common for operating systems to have tens of millions of lines of code [7, 12], and, as a result, even fully patched operating systems are likely to have security vulnerabilities. Such kernel level vulnerabilities are more dangerous than conventional application vulnerabilities because once the kernel is compromised by one application, the application has effectively compromised the whole system. Even outside of kernel vulnerabilities, it’s hard for users to know installers aren’t surreptitiously loading keyloggers, rootkits or other malicious tools that operate at the OS level.

3.2 Security Guarantees

Imagine an application that manages the nuclear launch system for a country. Even if the user completely trusts the application code for this program, ultimately, without some sort of guarantees of the behavior of the OS that supports it, the user has no reason to believe that the application is executing correctly or even as it appears.

Conventional user applications rely on the operating system to initialize and manage their address spaces throughout their lifetimes; however, in doing so, applications place a significant amount of trust in the OS. A malicious operating system might map an application’s address space incorrectly or change the contents of application code or data contained within its memory. Additionally, since the OS has access to both a program’s memory and registers, it might take advantage of this access to redirect control flow arbitrarily. A user running a the nuclear launch program might want to ensure that the operating system doesn’t redirect
control flow to a section of memory that fires the missiles instead of deactivating them.

Even with a system that provides memory isolation, many applications are vulnerable because they store configuration and user data files on disk. On a system without any guarantees of the integrity or confidentiality of such files, a malicious operating system might corrupt these files or change their contents such that an application is launched in an insecure manner.

Further, even on a system that provides secure storage and memory isolation, neither the user nor the applications that they communicate with through conventional channels on an untrusted operating system can trust the data they receive. A compromised operating system might have a keylogger installed that eavesdrops on user input and divulges the nuclear codes to a third party. A system that protects an application on these attack surfaces should provide some sense of mutual authentication of the user and trusted application as well as confidentiality and integrity for user I/O.

A creative manner to remove some degree of trust from the operating system is to design a system that verifies its behavior. InkTag, the work we build off of in this paper, provides mechanisms for validating some essential functions of an operating system.

### 3.3 Threat Model

We assume that the operating system is completely untrusted, but that we can trust all the hardware that runs below it. The attacker in our model wields the full powers of the host operating system and can resultingly try to modify application page tables, read user memory and execute any code. An attacker can drop and manipulate data sent through the OS such as user I/O. We also address a new class of attacks called Iago attacks [1] in which the attacker can maliciously craft the return values of a system call to try and cause harm to an application.

Our model places complete trust in the user application, and we cannot protect an application that directly leaks secrets by placing sensitive data in system call parameters or leaking user data in other manners. However, unlike previous work [10, 9, 3], our system
does not require application programmers to directly specify which application data should be secure and protects all data associated with a secure application.

4 Background

In this section, I discuss parts of the technical design of the initial version of InkTag and improvements in Sego, a newer version of the platform. I also explore why we need a specialized mechanism for secure user I/O on top of InkTag/Sego. I will not point out the entire scope of features provided by InkTag: a more in depth summary is available in the original paper.

System Architecture

The design of InkTag assumes that the operating system is completely untrusted. The trusted computing base consists of the hypervisor, which in InkTag is a modified version of KVM. Our slight modifications to the hypervisor enable it to verify the behavior of the untrusted guest OS. Although KVM runs its own operating system, it would be reasonable to implement the desired functionality in a more lightweight hypervisor, also known as a type 1 hypervisor.

Secure Processes

InkTag presents the abstraction of a trusted, high-assurance process, henceforth referred
to as a HAP. InkTag protects a HAPs context (registers), control flow and address space from the untrusted operating system. These guarantees are not provided to HAPs that are insecure as a result of application level bugs or programmer error. InkTag also does not provide any availability guarantees: for example, a malicious operating system might impede progress by never scheduling a HAP. Normal programs that do not need to take advantage of InkTags security guarantees may run unmodified, but HAPs must be compiled with libinktag, a small library that includes the code necessary for HAP communication with the hypervisor and system call interposition. An important distinction of InkTag from other systems [10, 9, 3] is that no application code needs to be modified to convert a normal program to a HAP.

Protecting Application Memory

InkTag introduces the S-page abstraction of pages of virtual memory. There are two views of an S-page, a plaintext view as visible to a HAP and an encrypted+hashed view as visible to the untrusted host. Each S-page has associated metadata, including the hash of the contents of the page, that must be kept up to date by the hypervisor. This metadata is maintained completely by the hypervisor and is invisible to the OS.

To protect the address space of a HAP, InkTag treats pages of a HAPs virtual memory as S-pages. When the operating system attempts to read from a HAPs address space, the hypervisor steps in and encrypts+hashes the page. When the HAP attempts to read a page of its own memory that is encrypted+hashed, the hypervisor decrypts and verifies the integrity of the page. This mechanism provides HAPs with guarantees of confidentiality and integrity for their address space.

Secure File I/O

InkTag uses this same functionality to allow HAPs to securely interact with files, even though file data is written to and read from disk by the untrusted operating system. InkTag refers to a group of S-pages and their associated metadata, which in some cases can be logically considered a file, as an object. Each object has a unique, 64-bit OID and the
hypervisor has a mechanism for mapping OIDs to filenames. The InkTag hypervisor encrypts and hashes pages of an mmaped file before it is written and decrypts them when they are read by a HAP. It’s important to note that, in InkTag, the untrusted OS is responsible for reading in S-pages from memory and also writing them out to disk, albeit in their encrypted form.

Access Control

InkTag also provides an interface for secure access control. Associated with the metadata for each file in InkTag is a set of formulas that specify the access permissions of different entities. InkTag relies on a system of attributes, which might specify users, groups or other principals. When a user invokes a HAP, they can provide it with any of their attributes, and the attributes associated with a HAP are evaluated against the access control formulas associated with the files. An example of a formula might be \( W = (\user.josh) \lor (\group.student) \), which would give write access to any HAP running with either of these attributes. InkTag’s system for delegating attributes also includes functionality to append attributes in a hierarchical manner to restrict the access of certain HAPs. A user with the attribute .user.sam might create and assign .user.sam.email to his email program to only allow his email HAP access to his email and not all of his files.

Paraverification

One of the core features of InkTag is paraverification, wherein the hypervisor verifies the behavior of key functions of the untrusted operating system. This requires the operating system to be slightly modified to inform the hypervisor of changes to a HAP’s state such as page table updates. An example scenario is one in which a HAP calls fork() and the hypervisor verifies that the operating system correctly maps the child’s address space. HAPs running on top of the operating system use hypercalls to communicate application intent to the hypervisor, and the hypervisor then correlates OS updates with application intent to verify an OS is not behaving maliciously. If the operating system fails to notify the hypervisor of important state changes or otherwise appears to be acting maliciously, the
hypervisor kills it.

HAPs in InkTag communicate with the hypervisor using hypercalls. A HAP is required to communicate the layout of its address space to the hypervisor on startup. The hypervisor keeps track of a HAPs address space as it evolves and uses information received from the OS and the HAP to verify the behavior of OS updates to a HAPs address space.

As a result of its design, InkTag addresses a new class of vulnerabilities known as Iago attacks [1]. In an Iago attack, a malicious operating system tricks an application into behaving in an undesired manner by choosing specific return values for system calls. An example of an Iago attack might be a call to `mmap()` in which the operating system decides to return an address in a section already contained in a processs address space. Such a scenario might cause a process to overwrite important data. To address these types of address space exploits, HAPs in InkTag maintain a linked list sorted by address that specifies entries in their address space. When the untrusted OS returns from a system call with a new mapping, it is required to provide an index in this linked list where the new mapping should be inserted to prove and let the application verify its intent.

**Control Flow Integrity**

To protect the control flow of HAPs, the hypervisor intercedes on all context switches involving HAPs. Upon a context switch, the hypervisor steps in to store a HAPs context in a secure region and then zeroes the registers. Part of libinktag, as compiled into each HAP, includes untrusted trampoline code that invokes the trusted hypervisor. When the OS schedules a HAP to run, it transfers control to this code and the hypervisor restores the processs state.

### 4.1 Improving InkTag

As discussed previously, one of the core ways InkTag provides address space integrity and privacy is by encrypting+hashing S-pages when data is written out or the OS touches an S-page. Although encryption and hashing requires some overhead, this can mostly be remedied
with hardware accelerated implementations such as aes-ni.

A more significant overhead is involved in maintaining an up to date digest/hash for S-pages, especially for S-pages that are part of secure files. This becomes even more difficult with attempts to provide crash consistency because a digest must be written atomically with the corresponding data. Its non-trivial to decide how to flush the contents of a secure file to disk while it is still being modified by a HAP. Is it best to pause the write in the application, encrypt+hash and block on the write to disk? Such decisions can have significant impacts on I/O and execution times.

**Revisiting Address Space Protection**

Sego improves on the designs of InkTag by making the untrusted operating system S-page aware. Unlike on Inktag, an untrusted operating system running on Sego is not allowed to touch S-pages in any way after they are reserved for secure data. We extend the boundary of the trusted computing base to include the disk controller. If the OS touches (attempts to read or modify) any S-page, the hypervisor immediately kills the operating system. This design has the advantage of removing the responsibility of the hypervisor to encrypt+hash S-pages, removing the need to maintain and update digests; however, the hypervisor is now responsible for several other low level functions involving S-pages.

Sego requires a slight modification to the Linux `struct page` on the OS to add a field to mark whether a page is secure. Whenever the untrusted OS does an operation that would involve touching the contents of an applications memory, it first checks this field on the `struct page`, and if the page is flagged as secure, it hands off the operation to the hypervisor via a hypercall. If the page is not marked as secure, the OS completes the operation normally. Example operations that need to be passed to the hypervisor for S-pages include copying pages for copy-on-write fork and zeroing pages for anonymous memory mappings.

**Moving the Trust Boundary for File I/O**

Sego also improves file I/O, providing an InkTag disk controller that handles all disk requests. Before reading a sector from disk, the untrusted OS checks against a bitmap to
see if the sector being read contains an S-page, and if so, it invokes a secure read. During a secure read, the InkTag disk controller will verify if the page being read is an S-page, and if so it will only read the contents of the S-page to a secure struct page. Such functionality requires the OS to reserve pages as secure before S-pages are read in from disk. In addition, the InkTag disk controller will refuse to read insecure data from disk to a secure struct page. Because all disk related I/O is handled by the disk controller, no data on disk needs to be encrypted or hashed. The InkTag driver also guarantees that S-page metadata, which might include information about OIDs or access control restrictions, is written atomically with data, providing crash consistency.

Secure User I/O

An important piece that is missing from the core of both InkTag and Sego is a mechanism for secure user I/O. Such functionality is necessary because paths for conventional user I/O send data across an untrusted channel directly through the OS. I will discuss the design of a mechanism for secure user I/O in the next section.

5 Design

Overview and Goals

Conventional user I/O relies on data passing through the operating system, which in InkTags model is an untrusted channel. An ideal abstraction for a trusted channel at the operating system level would provide confidentiality, message integrity and protection against replay attacks. The channel should also provide mutual authentication, so the user and HAP know with whom they are communicating. In addition, such a mechanism should prevent mutually distrusting HAPs from corrupting each others data. This interface should not add significant overhead to an applications performance and should require minimal, if any, modifications to application code. These channels should have comparable latency to their conventional, insecure counterparts.
Trusted Communications over Untrusted Channels

Establishing secure communications over an untrusted channel is a common challenge faced in networking today, frequently in the context of users accessing secure resources over untrusted network connections. Banks and other websites with sensitive material often address this scenario by supporting connections over HTTPS, which uses SSL to provide some of the guarantees that we outline above. SSH is another commonly used protocol, and although the protocols are similar, one of the key differences is in the increased amount of application level frameworks, such as the abstraction of multiple channels within a single connection, that SSH provides. Both protocols provide the abstraction of end to end secure communications over an insecure channel.

We considered designing our own protocol, and although a proprietary protocol design might have less overhead, there are several reasons we chose to use SSH over such an alternative. In the process of brainstorming use cases for our system, one scenario we imagined is a user working remotely from a trusted host interacting with multiple HAPs. The user might want to work with some graphical applications as well as some terminal based (textual) applications. Because SSH has integrated support for X, running graphical applications is greatly simplified. Also, SSH includes functionality for creating multiple channels inside a single session, which means we can have a client on the users trusted machine with one SSH session, and an inner channel for each HAP rather than many individual connections between the two. Choosing a standardized protocol makes the implementation significantly easier since we dont need to design and write a protocol from the ground up, but, more importantly, SSH has been tried and tested for over a decade. Lastly, because of the widespread use of SSH today, its traffic is generally permitted by most firewalls through port 22, whereas a protocol we write ourselves might be flagged and blocked as unrecognized traffic by organizational firewalls.

Authentication

An important part of designing a system that uses key based cryptographic protocol such as SSH is providing a mechanism for a user (or application) on one end of the connection
Figure 2: An Overview of Secure I/O

to authenticate itself to the entity on the other end of the connection. We take advantage of InkTags secure file and secure access control abstractions to provide a file system based public key infrastructure for authentication.

Proxying and Securing User I/O

In the context of a user remotely interacting securely with HAPs, we introduce the inmux and an outmux. When a user connects to their remote, untrusted host, they initiate a connection between the inmux, a process running on their trusted terminal, and the outmux, which is a HAP running on the remote host. A HAP that wants to initiate a connection for secure I/O with the remote user will first establish an SSH connection with the outmux. As each HAP initiates communication with the end user, the outmux verifies the identity of the HAP. The outmux then tunnels each HAPs connection as a channel to the inmux, which demultiplexes the per-application channels. On the users local machine, the inmux manages the display and encryption/decryption of user input and output. In summary, for each HAP
that communicates with the user, our scheme maintains one connection between the HAP and the outmux and one channel between the outmux and the inmux. Unfortunately, as a result of this split-connection approach, our system does violate end to end semantics; however, this design is much simpler than maintaining separate SSH sessions for each HAP with the inmux.

In line with our aforementioned design goals, we wanted to implement the above modifications in a way that required no modifications to application code. We decided to create a library, libsvt, that we compile with HAPs and contains the code to interpose on the conventional means for user I/O in a way that is transparent to applications.

6 Implementation

the Protocol

In order to build a mechanism that provides authentication, message integrity, and privacy for user I/O and IPC we needed to consider the existing kernel infrastructure and the tools available.

As discussed previously, we decided to use SSH to provide the abstraction of a secure channel with message integrity and privacy. Our implementation uses LibSSH and some of the cryptographic function libraries from OpenSSL. InkTag applications and libraries used with these applications must be compiled with an InkTag version of the standard C library, and in the process of changing the compilation steps of applications and libraries, we often had the opportunity to explore the inner workings of their build systems.

the Inmux and Outmux

In the previous section, we introduced the concept of an inmux, which runs on a trusted machine and interfaces with the user, and an outmux, which runs as a HAP on the untrusted host and mediates communication between the inmux and other HAPs. To provide a mechanism for authentication, where the inmux can verify the outmux is actually a trusted outmux, we use public key cryptography where the inmux. On the untrusted host, we com-
bine InkTags abstractions for secure files and secure access control so that only the outmux can read its private key file. Other HAPs running on the untrusted host can verify the outmux in a similar manner by reading the outmux’s public key, which is readable by all HAPs. In turn, the outmux can verify the identity of the HAPs it communicates with by reading each HAP’s public key, where each HAP’s corresponding private key is only readable by the specified HAP. Therefore a user connecting from an inmux can transitively verify the identity of a HAP.

Adding Secure I/O to Applications

In our design goals for InkTag, we strived to build a system that requires minimal modifications to applications. In creating our system for secure user I/O, we tried to maintain this constraint. In addition to writing the inmux and outmux, we created an additional library that is compiled with applications. To establish trusted channels for secure user I/O, our library initializes a connection to the outmux and, for terminal based HAPs, redirects stdin and stdout across this connection. For HAPs that use X-windows, we interpose on the call to XOpenDisplay and create a channel across the outmux+inmux that communicates with the X server on the end users machine.

An important part of designing any new system is evaluating its effectiveness. To evaluate the performance of secure user I/O we measured the latency of X events between a HAP and the inmux as well as that of terminal (text) based applications.

7 Evaluation

Developing and implementing the above outlined system for secure user I/O was a team effort, and I contributed in several ways to the projects development.

One of the key features of our system is that we reuse the SSH protocol to provide the abstraction of a trusted channel for communication. I was personally responsible for converting LibSSH to a version that can run on top of InkTag/Sego. This involved understanding and modifying the build systems for LibSSH and OpenSSL to link against the Sego/InkTag
<table>
<thead>
<tr>
<th>Application Type</th>
<th>non-HAP</th>
<th>Secure</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual</td>
<td>20000</td>
<td>1395</td>
<td>13×</td>
</tr>
<tr>
<td>Graphical (X11)</td>
<td>20100</td>
<td>1598</td>
<td>11×</td>
</tr>
</tbody>
</table>

Table 1: Roundtrip latency (in microseconds) for secure user I/O v1.

C library. I also tested the converted versions of the libraries to verify their correct functionality, and in a few cases had to remove some pieces, for example code that used system calls that are not supported by our system. In the process of converting LibSSH to a Sego compatible version, I discovered a bug in how LibSSH handles allocating buffers as it decrypts handshake packets. Specifically, LibSSH expects that `malloc(0)` will return a valid pointer, which is the behavior in applications compiled with the `gLibc` C library. `uClibc`, the C-library we use with applications written for Sego, returns NULL for calls to `malloc(0)`, which is also acceptable behavior per the official POSIX specification for `malloc`. This bug surfaced in a sample SSH application that I was testing that dropped the connection during authentication with a non-specific “authentication error”, and, after some investigation, I was able to trace the bug to its source. Subsequently, Alan Dunn, a co-author on the Sego paper, and I submitted a patch that was later accepted to the LibSSH open source community [4].

Initial Results

A more significant part of my role was in analyzing and understanding the performance of our system for secure user input and output. I initially wrote a suite of tests that measures the latency of terminal and X11 based applications compared to their conventional counterparts. One of the first tests that I wrote compared the roundtrip latency of repeatedly sending a byte back and forth between an application and the inmux. To evaluate the performance of a conventional system, I wrote a test that does so between a TCP server and TCP client over an SSH connection. As an analog for graphical applications, I designed a simple X11 application that repeatedly queries the remote X server (running on the user’s local machine) for the location of the mouse pointer and measures the time between when the query is requested and when a reply is received. Both of the latency benchmarks executed
100 queries, and I restarted the Sego VM between runs to try and reduce the impacts of any external variables. My initial results indicated that roundtrip latency over a conventional channel was about 1400 microseconds, and that roundtrip latency over our system was a staggering 20,000 microseconds.

Understanding the Results

In an attempt to determine the source of this significant overhead, I instrumented various parts of the system to figure out where this latency was occurring. My initial tests measured the latency as visible to the application, so the next logical step was to measure the latency as visible to the outmux and inmux. I inserted timing mechanisms in the outmux and measured the average time between when a packet was received from the socket connected to the HAP and when the response packet correlated to the initial packet was sent back to the HAP. The purpose of this test was determine what portion, if any, of the overhead could be associated with the link between the outmux and a HAP. The roundtrip time perceived at the outmux seemed to be approximately 19,800 microseconds, indicating that the link between the HAP and the outmux was not the source of the majority of the overhead.

My next step was to further instrument the outmux, and I decided to measure the total time a packet spent in the outmux. To do so, I recorded the time at which a packet was read from the socket from the HAP and the time at which the corresponding packet was written to the socket going to the inmux. I also calculated the time on the reverse path, from when the outmux received a packet from the inmux to when the corresponding packet was written to the inmux socket. I determined that the overhead incurred in both of these operations was at most 100-200 microseconds each. I wrote similar tests on the inmux with similar results. These results led me to believe that the overhead was not incurred in either the inmux or the outmux.

Analyzing the results of all the tests I had run up to this point, it seemed like most of the overhead was occurring somewhere between the inmux and the outmux. I tried to print and correlate the timestamps as packets were sent from the outmux and received at the inmux, but due to clock drift, the timestamps across the machines were not comparable (even after
running NTP, the system times were not close enough to compare at this granularity). I tried
to remove any network latency that might have been affecting the benchmark by running
the inmux inside the Sego VM, but this increased the overall latency, likely as a result of
scheduling overhead.

Subsequently I decided to consider any overheads that might exist between the outmux
and the inmux. To do so I had to fully understand the path between the outmux and inmux.
In our initial design, when the user starts the inmux on their local machine, it execs an
instance of SSH and creates a PTY (pseudoterminal channel) on which it communicates
with the local SSH client. This SSH client creates an SSH session with sshd on the remote
machine and instructs sshd to start the outmux on the remote machine. Upon receiving the
connection and request to start the outmux, sshd sets up a PTY, sending data it receives
from the ssh client to stdin on the outmux across the PTY. The outmux communicates with
the inmux by writing to its stdin file descriptor, and this data is sent to through the PTY
to the remote SSH client and ultimately to the inmux. The inmux and outmux create an
SSH session over this path to provide authentication, integrity protection and confidentiality
from the untrusted sshd and OS, which controls the PTY.

Secure I/O v2

After gaining a more thorough understanding of the connection between the outmux and
the inmux, I considered which parts of this architecture might have significant impact on
the latency along this path. The separate PTY between the inmux and ssh running on the
user’s local machine seemed like it could be contributing part of the overhead, so I decided
to build an ssh client into the inmux. Removing the PTY from between the inmux and ssh
would also allow me to remove the PTY from between the outmux and sshd.

Many facets of the design of our mechanism for secure user I/O rely on having a second,
trusted SSH session running inside a conventional, untrusted SSH session that is established
between the local host and sshd running on the remote host. The outer SSH session is
untrusted because we don’t trust sshd on the remote machine. The SSH protocol has two
types of connections: SSH sessions and SSH channels. SSH sessions are established between
hosts and various types of application data are sent inside of different channels within these sessions. All the channels inside a session share a key. Our initial system design relies on ssh on the user’s machine to establish a session with sshd and then create a channel inside this session that communicates with the outmux through the remote PTY. We establish the inner, trusted SSH session across this inner channel between the inmux and the HAP.

Improving LibSSH

To build an ssh client into the inmux, I needed to directly create an SSH session, an SSH channel inside this session and then an inner SSH session inside this channel. This was not as much of an issue when using a separate SSH client because the SSH client communicates over a PTY with the inmux, and the inmux can treat the file descriptor from this PTY as a two way socket over which it can establish a new session. However, when I built the outer SSH session and channel into the inmux, I had to figure out how to create a new SSH session over a raw SSH channel, without any PTY between the two. Although LibSSH provides a mechanism to get a two way file descriptor for an SSH session, there is no similar mechanism for a single channel because of the way that SSH channels are multiplexed in a session. To
provide the abstraction of a file descriptor that I could treat as a socket from a channel, I had the outmux fork a child thread that initializes and polls this channel, writing any data it receives to a pipe that the parent reads. The child thread polls on another pipe that the parent writes to when it wants to send data. The parent establishes a nested SSH session across these pipes and through the channel.

To use separate file descriptors for input and output for an SSH session, I needed to modify LibSSH to expose such an interface. There was some degree of implementation of this functionality behind the scenes, but this functionality was neither publicly exposed, nor, as I soon learned, well tested. After making what seemed to be the necessary modifications, I tried to test the system. After some minor debugging, I was able to establish the outer and inner sessions between the inmux and outmux without issue; however, strangely enough, once an application was launched an a channel was created within the inner session, I was able to read data, but data I wrote to the channel didn’t seem to ever get written to the pipe. Upon investigation, it appeared that the socket callback that was responsible for flushing the channel buffer to the output socket was not getting called. I was able to trace the call to `ssh_channel_write()` to the inner channel to a point where the POLLOUT flags were set on the outgoing file descriptor, but `poll()` didn’t seemed to be reacting or noticing the flags.

I was eventually able to trace the bug to a function call to `ssh_events_add_session()` that was responsible for copying the existing file descriptors for a session to a new polling context. This function includes a for loop that iterates across the existing poll file descriptors, calls `ssh_poll_ctx_remove()` on each descriptor and then `ssh_poll_ctx_add()` to add them to the new context. This design seemed reasonable until I looked into the code for `ssh_poll_ctx_remove()`. It turns out that this function removes a file descriptor from its location in an array, decrements the number of used file descriptors and then, to be space efficient, copies the last file descriptor in the array to the location of the file descriptor that was removed. Because `ssh_events_add_session()` iterates over the number of used file descriptors, a quantity that decreases as each file descriptor is removed, the first file descriptor is removed, the index in the for loop is then 1 and the length is 1, so the loop to copy the file descriptors terminates,
leaving the file descriptor for channel writes in the old array that is no longer used. This code path was likely not used before because most applications will have at maximum one file descriptor that is used at the time the function is called, where we have two because we are using one for input and one for output.

**Evaluating Secure I/O v2**

After making the required modifications to LibSSH and to the code we use for connecting the inmux to the outmux, I was able to fully remove the PTY and the ssh client from the user side of the connection. Unfortunately, much to my disappointment, the latency overhead didn’t change and I was still seeing approximately a 20,000 microsecond roundtrip latency.

The next modification that I decided to make was to remove the PTY from between `sshd` and the outmux. To do so, I had to modify the outmux code so that it communicated over both stdin and stdout with the inmux rather than treating stdin as bidirectional file descriptor. This fairly minimal modification, which would not have been possible without all of my previous changes, immediately resulted in a significant drop in latency overhead to
Table 2: Roundtrip latency (in microseconds) for secure user I/O v2.

<table>
<thead>
<tr>
<th>Application Type</th>
<th>non-HAP</th>
<th>Secure</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual</td>
<td>664</td>
<td>849</td>
<td>28%</td>
</tr>
<tr>
<td>Graphical (X11)</td>
<td>760</td>
<td>900</td>
<td>18%</td>
</tr>
</tbody>
</table>

where our system now incurred only an 28% latency compared to the conventional system. Although it’s not immediately clear what caused this overhead, it’s likely something related to how OpenSSH handles TTY channels compared to stdin/stdout channels.

In the future I hope to submit a patch to the LibSSH community to expose the facilities for using separate file descriptors for input and output as well as a patch for the incorrect function for copying poll file descriptors. In the mean time, my analysis and modifications, which were about 1400 lines in total, of our mechanism for secure user input and output has resulted in a newer version of the system that significantly outperforms the previous one.

8 Related Work

8.1 Using VMs to Isolate Applications

One of the earliest approaches to minimizing the TCB for applications was to isolate applications in their own virtual machines. Garfinkel et. al [5] propose a system wherein each application runs on its own customizable software stack on top of a trusted virtual machine manager. The system provides an attestation mechanism to allow users of the application to verify the identity of the VM and hardware based on a certificate chain of trust originating from a per-device private key stored in hardware. The authors suggest using an encrypted channel from user devices to the trusted virtual machine manager to prevent malicious drivers on the host from interfering or eavesdropping on user communication. Although this system certainly protects from operating system vulnerabilities that are a result of other applications through isolation across virtual machines, it does not protect against a malicious OS or a VM image that was provided to the user initially in a compromised manner.
The authors of Cloud Terminal [8] highlight the fact that many of the security vulnerabilities that plague users today may be the result of the use of operating systems that are not fully patched or not well audited. The authors propose separating the architecture for running applications into two parts: a secure thin terminal that runs in an isolated, secure manner on the client side and a cloud rendering engine that runs the actual user application and is maintained by the application publisher. On the remote server side, each application session is loaded in its own virtual machine, allowing the developer to prevent malicious users from affecting other sessions. On the client side, a thin microvisor is installed below the operating system that can encrypt user I/O when the client is using a secure remote application. When a user wants to start a secure application, they open the secure thin terminal application that runs on top of the microvisor and establish a connection with the cloud host of their application. The microvisor is able to intercept user input while the user interacting with the secure terminal and send it over an encrypted connection to the cloud rendering engine. In turn, the microvisor writes output directly to the appropriate channels, preventing the guest OS from intercepting or reading its data. The remote server is able to verify the identity of the microvisor and thin client using attestation enabled by the use of a TPM and hardware virtualization technologies. Although this system certainly provides a mechanism for secure user I/O, it does not protect the application on the server side from its own operating system acting in a malicious manner. On the other hand, this architecture does not seem far reached as more applications move to cloud hosted models.

8.2 (Simulating) Hardware Isolation

Many authors have proposed taking advantage of specialized hardware to isolate applications from the operating system. Lie et. al [6] design the XOM processor and XOMOS operating system to provide applications with a secure compartment abstraction with which they are isolated from the operating system and all hardware outside of the CPU. The authors introduce the concept of execute only memory, which they present as a mechanism for copy protection and tamper resistance for secure applications. All data that corresponds
to a secure application (a compartment) is encrypted whenever it is outside of the CPU, ultimately reducing the TCB to only the CPU. Creating an application that takes advantage of the security benefits of XOMOS requires significant modification on the part of the application programmer who must write a large part of the software stack of their program. XOMOS does not provide a mechanism for secure user I/O. McCune et. al propose Flicker [10], a system that protects pieces of applications from malicious operating systems as well as much of the hardware stack running beneath them by taking advantage of CPU hardware advancements including TXT/SEM and TPMs. Building on top of these technologies, application developers can isolate parts of their applications into secure pieces of application logic (PALs) that run in a different context and are completely isolated from the OS, with only a 250 line TCB. Remote hosts can verify the execution of trusted code using attestation features of TPMs, which can create a verifiable cryptographic hash of executed binaries as well as their inputs and outputs. To provide a channel for secure communication with a remote host, Flickers authors propose creating an asymmetric key pair using a secure PAL and using the public key to initiate a secure channel from the remote side. Because PALs are run completely outside of the OS, application writers are responsible for implementing almost the entire stack in these pieces of code. Further, application writers must be able to identify each part of the application that should be executed securely, which is likely a non-trivial task. Finally, Flicker also incurs a significant performance overhead because each time a PAL is executed the CPU switches completely out of the OSs context. McCune et. al present TrustVisor [9] in an attempt to address the performance downfalls of Flicker by introducing a trusted hypervisor with a micro-TPM. The system uses a hardware TPM to attest to the identity and integrity of the trusted hypervisor before it is launched. Subsequently the trusted hypervisor can enforce the isolation of secure pieces of application logic and attest to their execution. The hypervisor ensures isolation by isolating secure application data from the OS and devices. Its possible for users to establish a secure channel on top of TrustVisor in the same way as they could using Flicker. As a result of the isolation features of TrustVisor, application programmers are still left with the somewhat daunting
task of isolating specific trusted regions of their programs.

### 8.3 Enforcing OS Behavior

In a precursor to InkTag, the authors of Overshadow [2] propose isolating applications from an untrusted OS by cloaking (encrypting) the contents of an applications memory when it is out of context. Overshadow uses a trusted hypervisor to encrypt and hash a secure applications memory and manage loading and saving a trusted applications context, very similar to the mechanisms implemented in InkTag. Overshadow does not persist an applications private key, meaning that any secure data written to disk is unavailable after the application is closed. Overshadow also does not provide a mechanism to verify an operating systems behavior, and a malicious operating system could trivially compromise a secure application by mapping pages incorrectly or crafting return values from system calls. Although the authors mention that a path for secure user I/O would be desirable, they do not implement or discuss the design for one in the paper.

In Virtual Ghost [3], Criswell et. al achieve application isolation without adding a supervisory layer beneath the operating system. The authors build on an existing secure virtual architecture which uses compiler instrumentation and run-time checks on OS code to ensure application isolation. Application developers can take advantage of a new type of application memory, ghost memory, that is isolated from other applications and the OS. Because only memory designated by applications as secure memory is isolated from the rest of the operating system, secure applications experience significantly less overhead. This idea of a developer determined boundary for secure data, similar to the isolation of PALs in Flicker/TrustVisor, puts significant responsibility on the application developer and is likely to result in the inadvertent leaking of secure data. The authors of this paper do not discuss secure user I/O.
8.4 a Secure Terminal Interface

McCune et. al present Bump in the Ether [11] as a secure terminal interface for user input. In their work, the authors highlight the insecurity of user I/O in contemporary systems, specifically citing that many window managers allow any application to subscribe to all keyboard events, even while out of focus. They propose the use of a trusted mobile device that allows users to choose the application that should receive their input. The mobile device interacts with a kernel module on the users machine across an encrypted channel. The kernel module uses hardware attestation to prove its identity. Although this system provides more fine grained security guarantees for user input, it does not provide any guarantees for user output. In Sego, we use a similar approach to tunneling input for each application to a trusted process on the host.
References


[8] Lorenzo Martignoni, Pongsin Poosankam, Matei Zaharia, Jun Han, Stephen McCamant, Dawn Song, Vern Paxson, Adrian Perrig, Scott Shenker, and Ion Stoica. Cloud terminal: secure access to sensitive applications from untrusted systems.

