SwitchMan: An Easy-to-Use Approach to Secure User Input and Output

Shengbao Zheng*, Zhengyu Zhou*, Heyi Tang† and Xiaowei Yang*
*Department of Computer Science, †Department of Computer Science
*Duke University, †Tsinghua University
*Durham, USA, †Beijing, China
*{szheng, zzy, xwy}@cs.duke.edu, †tangheyi.09@gmail.com

***Abstract*—Modern operating systems for personal computers (including Linux, MAC, and Windows) provide user-level APIs for an application to access the I/O paths of another application. This design facilitates information sharing between applications, enabling applications such as screenshots. However, it also enables user-level malware to log a user's keystrokes or scrape a user's screen output. In this work, we explore a design called SwitchMan to protect a user's I/O paths against user-level malware attacks. SwitchMan assigns each user with two accounts: a regular one for normal operations and a protected one for inputting and outputting sensitive data. Each user account runs under a separate virtual terminal. Malware running under a user's regular account cannot access sensitive input/output under a user's protected account. At the heart of SwitchMan lies a secure protocol that enables automatic account switching when an application requires sensitive input/output from a user. Our performance evaluation shows that SwitchMan adds acceptable performance overhead. Our security and usability analysis suggests that SwitchMan achieves a better tradeoff between security and usability than existing solutions.**

***Keywords*-Usability, Input/Output Security**

I. Introduction

An important goal of computer security is to protect private user data from unauthorized access or modification. In recent years, we have witnessed a wide range of new technologies to protect user data, including full disk encryption, secure data transmission protocols such as TLS/SSL, and secure cloud storage. However, sensitive user input/output data remain vulnerable to data stealing attacks by malware residing in a user's computer. Keyloggers can capture every keystroke of a user and screen scrapers can take screenshots of any displayed window. As an example, between 2013 and 2015, attackers used the Carbanak malware [1] to infect bank computers and then recorded videos of a victim's screen and keystrokes to obtain sensitive banking information. They successfully stole money from around 100 financial institutions and the total financial loss amounted to almost one billion dollars.

Protecting sensitive user input/output is challenging because modern operating systems for personal computers (including Linux, MAC, or Windows) provide user-level APIs for one application to share the I/Os with another application running with the same user identifier. For example, X11 provides a function called `XGrabKeyboard()` to allow an application to capture the keyboard events of another application. Once a client application connects to an X server, it would share its I/O paths with all clients connected to the same X server. These function calls allow any malware to steal the keyboard input and screen output of a user's applications.

Previous proposals to address this challenge fall into four broad categories. Work in the first category protects a user's I/O paths at the hardware level. An example is the Intel Protected Transaction Display (PTD) solution [2]. Work in the second category proposes to use a second mobile device as a trusted input/output device [3], [4], [5], [6], [7], [8]. Work in the third category uses virtual machines to isolate trusted applications [9], [10]. Finally, work in the fourth category proposes to enhance an OS by implementing fine-grained access control for I/O interfaces so that one application cannot access another application's I/O paths by default [11], [12], [13], [14].

Each of the previous solutions requires a unique trusted computing base (TCBs). However, they all require significant user management to achieve the desired level of security. A user needs to decide which data are sensitive and then switch to a trusted hardware (e.g. a mobile device) or a trusted terminal (e.g., one runs inside a trusted virtual machine) or both to input/output sensitive data. We hypothesize that it is challenging for a non-expert user to manage these tasks. Therefore, it is beneficial to explore a design alternative that can automatically manage the switching to sensitive data input/output without user involvement.

In this work, we propose SwitchMan, an architecture that enables a server to switch a user to a secure terminal for sensitive user input/output. At the heart of SwitchMan lies a protocol that enables a remote server (e.g. a web server) to embed a secure terminal switching request inside its traffic stream even if the client's software (e.g. a browser) is untrusted (§ IV-C). The TCB running on the client will intercept the request and switch the user to a secure terminal.

The SwitchMan architecture can support different TCBs. One design choice is to run a secure terminal inside a trusted VM, and let the VM manager (VMM) intercept the switching request. As a preliminary step, in this paper, we assume that the OS kernel and its graphical interface are trusted. We use the OS's built-in user-level separation

DOI 10.1109/SPW.2019.00029

mechanism to create a trusted input/output terminal. This preliminary design allows us to quickly prototype SwitchMan, and evaluate its performance. It is our future work to study how to reduce SwitchMan's TCB.

In the preliminary SwitchMan design, the OS provides each user with two accounts: a protected account and a regular account. The protected account is a "commodity" account such that it only runs a small set of applications trusted by the OS vendor. A user cannot install arbitrary applications under this account and can only use it as a trusted terminal, i.e., inputting and outputting sensitive data. A user's regular account is the same as the account a user has today. He can use it without the above mentioned constraints.

We have implemented the SwitchMan design using Linux and evaluated its performance. We have also conducted a preliminary analysis of its security and usability (§ VI). Our performance evaluation shows that SwitchMan adds low overhead to the existing system. Our usability analysis suggests that SwitchMan is easier to use than previous proposals.

We make two main contributions in this work. First, we introduce the SwitchMan architecture as an easy-to-use alternative to secure user input/output. Second, we build a SwitchMan prototype using Linux and evaluate its performance. Our analysis shows that it improves personal computers' security compared to the status quo and is easy to use. Our performance evaluation shows that SwitchMan has low overhead.

II. Related Work

In this section, we describe the related work.

VM-based isolation: A large body of work proposes to use virtual machines to separate untrusted applications from trusted ones [9], [10], [15], [16], [17]. VM-based isolation offers strong security but it requires significant user management effort. A user must choose the right VM for the right applications. Otherwise, untrusted applications may gain access to a user's sensitive data. The Android [17] architecture assigns each application a unique user identifier. This isolation model is similar to SwitchMan's. However, the Android architecture only allows an application to set its I/O sharing permission at the application level, not based on the sensitivity of its I/O data. An application can choose to allow I/O sharing or not. If it allows I/O sharing, then malware can steal its I/O data; if it does not, then a user cannot take a screenshot of the application. In contrast, SwitchMan allows a server to choose the level of I/O protection based on the sensitivity level of the data. The same application, e.g., a browser, can allow screenshots for non-sensitive data but prohibits I/O data sharing for sensitive data.

Device-level isolation: There also exist proposals that use a personal device such as a mobile phone to input/output sensitive user data [3], [5], [6], [7], [8], [18], [19]. These proposals assume that a user's personal device is trusted and use it for the input or output of sensitive data. We consider it inconvenient for users to use a separate device for sensitive input/output. In addition, smart phones can also be infected with malware and may no longer be trusted.

Securing the input/output path by minimizing TCB: Some researchers argue that obtaining input/output data does not require a general purpose OS and propose to use a separate module isolated from a user's OS to receive a user's input/output data. Borders et al. propose to use a small Trusted Input Proxy (TIP) [20] to receive secure user input. The Cloud Terminal architecture [21] proposes to run all application logics inside a server hosted at a cloud provider, including the graphic rendering logic. A user's computer only runs a small TCB to receive input from the user and render graphical output from an application running in the cloud. Bumpy [4] uses an encrypted keyboard and mouse to protect sensitive user input. Zhou et al. [22] propose to build a trusted I/O path between I/O devices and a user's trusted program.

All these solutions assume that the OS cannot be trusted and require a user to initiate the switching to a secure input/output device. They require a user to install additional software or require special hardware such as encrypted keyboards [4].

Fine-grained I/O control: Nitpicker [13] uses a minimized secure graphical user interface. SELinux [11], [12] proposes to enforce strict access control of an X server. DriverGuard [14] proposes to use fine-grained I/O flow protection in the kernel space. However, these solutions require significant changes on application program interfaces (APIs), operating systems, or window managers, and have not gained wide adoption.

Advanced hardware: Intel protected transaction display [2] proposes to use a random input pin pad to replace keyboard input to prevent keylogging. It provides hardware-level support to prevent an application from taking screenshots of the input pin pad. However, it has not been widely available on general purpose computers.

Compared to the existing solutions, SwitchMan has two main differences. First, it does not require user involvement to switch from an untrusted environment to a trusted one. Second, it does not require changes on client applications, and only requires a small set of changes on the OSes of a client and a server, and on the server software.

III. Overview

We describe SwitchMan's design goals, assumptions, and adversary model in this section.

A. Goals

Protecting sensitive input/output data against user-level malware. SwitchMan aims to secure sensitive input/output data from user-level malware running on a conventional multi-user OS such as Windows, MAC OS, or Linux. In this work, we only consider GUI and keyboard access, and leave how to extend the design to multi-modal interactions for future work. We do not aim to prevent data leakage when attackers compromise a user's OS or other software with root privilege. Local applications may store sensitive data on a computer's permanent storage. Securing access to such data is outside the scope of this work.

Easy to use. SwitchMan aims to be easy to use, as past experience suggests that it is challenging for non-expert users to adopt sophisticated or cumbersome security solutions [23].

Efficient. We aim to make SwitchMan efficient to use. SwitchMan should not introduce user perceivable performance degradation.

B. Assumptions

Trusting OS and its vendor. SwitchMan's design assumes that a user's OS kernel and the graphical system distributed with the OS can be trusted, as in previous work [3], [5].

We make this assumption mainly because of our design goals. Trusting the OS makes SwitchMan easy-to-use. By trusting the OS, we can provide a turnkey solution to the user. An OS vendor can distribute SwitchMan with the OS and turn it on by default. We expect that ease of use can increase the chance of user adoption.

Admittedly, trusting the OS has the drawback that when a user's OS is compromised, SwitchMan cannot secure the access to sensitive user input and output data. However, we believe there are a few remedies that can reduce the security risk of this assumption.

First, there exist techniques such as the Integrated Measurement Architecture (IMA) [24] that can measure and attest an OS's integrity. Second, trusting an OS significantly reduces the TCB compared to the status quo. Today, if one application residing in a user's account is compromised, the application can steal sensitive user input/output. We obtain data from the Common Vulnerabilities and Exposures (CVE) dataset [25] for the time period from 2013 to 2018. We find that the percentage of privilege escalation vulnerabilities and root privilege vulnerabilities among all vulnerabilities are in the range of $[3.01\%, 9.34\%]$ and $[0.17\%, 0.65\%]$ respectively. This result shows that the number of OS vulnerabilities is much fewer than the total number of vulnerabilities, suggesting that trusting the OS rather than all applications can significant reduce the security risk of data leakage.

Finally, there exists market competition among OS vendors. The OS vendors are accountable for security breaches caused by OS compromises, and accountability can motivate an OS vendor to improve its security, reducing the risk that the OS is compromised.

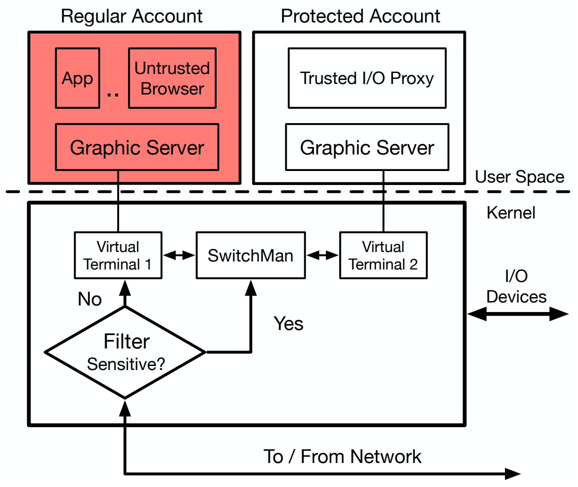

Figure 1. **This figure shows the overall SwitchMan architecture. Each user has a protected account and a regular account. A user switches to his protected account and uses the trusted I/O proxy running under that account for sensitive input/output. SwitchMan can help secure both sensitive local and network input/output data.**

Secure Storage & Network Transmission. We assume that sensitive user data can be securely transmitted by protocols such as HTTPS/TLS or SSH and can be securely stored on disk by file system encryption technologies such as [26].

C. Adversary Model

No Physical Access. We assume an attacker does not have physical access to a user's computer. Therefore, an attacker cannot capture a user's screen with a camera or capture a user's key strokes with a hardware keylogger.

Malicious Man-in-the-Middle (MITM). We assume that there are active attackers in the middle of the network who attempt to modify and access network traffic to further extract user sensitive data. We assume the malware residing in a user's computer and MITM may collude to attempt to steal user data.

IV. SWITCHMAN DESIGN

In this section, we describe how we design SwitchMan to achieve its design goals.

A. SwitchMan Architecture

Figure 1 shows the SwitchMan's architecture. A computer's OS assigns two user accounts to one user. One is a regular account, where the user has the freedom to run any application. The other is a protected account. This account comes with a set of pre-configured software that the OS manufacturer trusts. A main purpose of this account is to provide a trusted terminal for users to input/output sensitive data. Each user account has its own display server. Applications running under the regular account cannot connect to the protected account's display server. Each server uses its own virtual terminal so that their I/O paths are isolated at the software level.

The design of SwitchMan includes four main components: 1) a program called the Trusted I/O Proxy (TIOP) running under a user's protected account; 2) a kernel module called SwitchMan for managing the switching between a user's two accounts; 3) a kernel filter for managing sensitive network input/output data; and 4) a network protocol which we call SwitchMan's Network Protocol (SNP). SNP enables a remote server to send a request to a user's OS to switch the user to his protected account for accessing sensitive input/output data (§ IV-C). SwitchMan can secure both local and network input/output. Due to space limit, we only describe how to secure network input/output in this paper.

Next, we describe each design component in more detail.

B. Trusted Input/Output Proxy(TIOP)

In the SwitchMan design, a user interacts with sensitive data via TIOP. One can view TIOP as a simple web browser distributed by a user's OS vendor. It displays the sensitive output received from a remote server and takes a user's input. To be secure, the OS will prohibit a user from installing arbitrary extensions to TIOP and may also disable advanced browser features such as Javascript to reduce security risk.

Figure 2 shows a sample user experience flow when a user is using SwitchMan. A user does his normal operations in his regular account. When a server sends a switching account, a user is switched to using TIOP under his protected account. A user must know whether he is under his protected account to prevent a malicious program from impersonating TIOP. In the SwitchMan design, a user chooses a secret background image for his protected account when he creates his accounts. The image will be encrypted and stored with a user's other login credentials. A user will see this image when he is under his protected account. In the example of Figure 2, the white-check-mark-on-green-shield image is the user's chosen background image.

TIOP is the only application connected to the virtual terminal running under the protected account. Thus, other applications under a user's normal account cannot connect to the same protected virtual terminal to steal sensitive user input/output. A user is switched back to his regular account after he finishes sensitive input/output.

C. SwitchMan's Network Protocol (SNP)

SwitchMan's design includes an HTTPS-based protocol which we refer to as SwitchMan's Network Protocol (SNP). SNP enables a remote server to securely request the SwitchMan OS to switch a user to his protected account. We choose to base the design on HTTPS due to its prevalence. However, we believe SNP can be adapted to other TCP-based protocols.

SNP has three main features. First, it is backward compatible with the present TLS/TCP protocol stack. A non-SwitchMan-upgraded client can continue to connect to a SwitchMan-upgraded server and vice versa. Second, the protocol is resistant to MITM attacks. A MITM attacker can at most launch denial of service attacks by discarding traffic, but cannot steal a user or a server's sensitive data. Third, it does not require the client application (e.g. a browser) with which the server interacts to be trusted. A malicious browser is treated the same way as an MITM. That is, it can at most discard a server's request to switch a user to a protected account, but cannot steal sensitive input/output data.

Next we describe how SNP works as shown in Figure 3.

1) Step 1: TCP/TLS handshake: In the first step, a client connects to a server via HTTPS. This initial step is the same as the standard TCP/TLS protocol except that 1) a SwitchMan-enabled client and server will exchange additional information via TCP options and 2) a client's OS will intercept and store a server's SSL/TLS certificate, as shown in pseudo code SNP Step 1. We introduce a new TCP

```
SNP Step 1 :
// TCP Handshake.
$ClientOS \rightarrow Server : SYN$ w/ $Opt(SM)$
$ClientOS \leftarrow Server : SYNACK$ w/ $Opt(SM\_Echo)$
$ClientOS \rightarrow Server : ACK$
// TLS Handshake.
$Browser \rightarrow Server : Hello$
$Browser \leftarrow Server : Hello + Certificate$
$Browser \rightarrow Server : Key$
```

option SM for backward compatibility. If a client does not include this option in its initial handshake with the server, it indicates the client is not SwitchMan-enabled. Similarly, if the server does not echo back the TCP option SM_Echo, it indicates the server is not SwitchMan enabled. When either of these happens, SNP falls back to the standard HTTPS. During this step, the client OS will also intercept the server's SSL/TLS certificate for later verification purpose. We note that a malicious browser or malware with user-privilege cannot intercept a TCP option, because it is the OS that receives and processes a TCP option and the OS can decide not to pass it to an application. Therefore, malware cannot degrade a SwitchMan-enabled client/server to stop running SNP.

2) Step 2: Server Initiated Switching: After a client and a server have established an HTTPS connection, they may proceed with their normal data exchange. When the server desires to receive or send sensitive data to the client, it will send a switching request to the client. When a client OS receives a server's switching request, it needs to 1) validate the request is from the server, and 2) invoke the

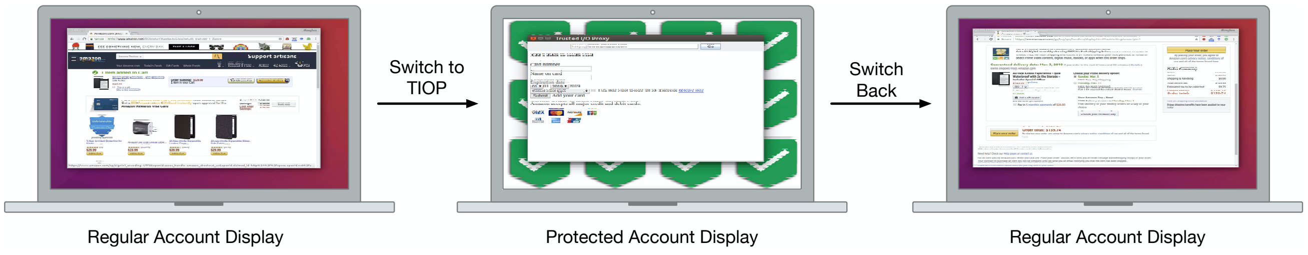

Figure 2. **A sample user experience flow of switching between two accounts. The leftmost figure shows a user accesses an e-commerce website under his regular account. The middle figure shows the user is switched to using TIOP to enter his credit card information under the protected account. The protected account display has a secret background image. The rightmost figure shows that the user is switched back to his regular account.**

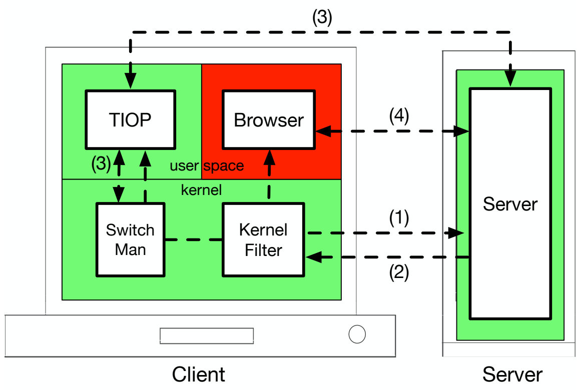

Figure 3. **This figure shows how SwitchMan's network protocol works at a high level. 1) A client connects a server via HTTPS. 2) If a server desires to receive sensitive data or display sensitive data, it notifies the client's kernel. The client's OS intercepts this signal, and switches an eligible server session to a user's protected account. 3) The user uses the trusted TIOP program to interact with the server. 4) The user resumes his previous session in his regular account after the TIOP session finishes.**

TIOP program from a user's protected account to exchange sensitive data with the server. To accomplish the first task, we let the server sign its request. Since in Step 1, the client OS intercepts the server's TLS certificate. It can validate the authenticity of the server request with that certificate.

It is challenging to accomplish the second task, because TIOP must establish a secure connection with the server, authenticate itself to the server, and associate its new connection with the existing connection established in Step 1. Otherwise, when the server receives the new connection from the TIOP program, it cannot validate whether the client is the one it interacts with previously, and does not know what sensitive data to send to or receive from the client.

One can use a secret session identifier to establish the association between the client/server's original connection and the new connection TIOP initiates. The server generates a secret identifier, and uses it to uniquely identify its request, and sends it only to the intended client.

However, how to send this secret session identifier securely becomes a design challenge. In the SwitchMan design, the communication channel between a server and a client is either a TCP option field, or an HTTPS connection. If the server sends the secret session identifier to the client via a TCP option, it is not encrypted. A MITM may intercept this session identifier and impersonate the TIOP program to establish a connection with the server. If the server sends it encrypted in the HTTPS payload, an untrusted application will receive it and may impersonate the TIOP program.

We address this challenge by splitting the secret session identifier into two halves. The server sends the first half of the session identifier as a TCP option and the second half in the HTTPS payload to the client. The client's OS intercepts the first half, and the untrusted application (e.g., a browser) receives the second half. The server signs the second half and includes the signature in the HTTPS payload to prevent an untrusted application from tampering the session identifier. The untrusted application stores the second half at a well known location. The TIOP program cam read it from the well known location, validate a server's signature, combine the two halves into a secret session identifier, and establish an HTTPS connection with the intended server. TIOP will include the secret session identifier in the payload of its HTTPS connection to authenticate itself to the server and associate its new connection with the previous connection between the untrusted client program and the server. The pseudo-code SNP Step 2 illustrates this design.

```
SNP Step 2 :
$Browser \rightarrow Server : Request$
$Browser \leftarrow Server : Normal\_Data$
// First half of the switching request.
$TIOP \leftarrow Server : TCP\_Option(nonce_1, nonce_{id})$
// Second half of the switching request.
$Browser \leftarrow Server :$
$https(JS(URL_{sensitive}, nonce_2, nonce_{id}, signature))$
```

The $nonce_1$ and $nonce_{id}$ fields constitute the first half of the secret session identifier, and $nonce_2$ and $nonce_{id}$ constitute the second half of the secret session identifier.

SwitchMan design assumes the untrusted application is a browser. So the server will send the second half of the session identifier in a Javascript with code to store it at a well known location. In the SwitchMan design, a user can read down from a protected account to his regular account. So the TIOP program can read any file located inside a user's regular account.

3) Step 3: TIOP Connects to the Server: In SNP Step 3, TIOP connects to the sensitive URL ($URL_{sensitive}$) sent by the server. The TIOP program authenticates itself by presenting the secret session identifier composed by $nonce_1$, $nonce_2$, and $nonce_{id}$. Note that a malicious application cannot modify $URL_{sensitive}$ because it is protected by the server's signature in Step 2.

SNP Step 3 :

$TIOP \rightarrow Server$:
$https(URL_{sensitive}, nonce_1, nonce_2, nonce\_id)$
$TIOP \leftrightarrow Server : Sensitive\_Data$

With this design, a MITM cannot intercept the secret session identifier as it is protected by the HTTPS connection. In the meantime, the untrusted application cannot impersonate TIOP either, because it does not have the first half of the secret session identifier.

4) Step 4: Switching back to the regular account: Finally, when TIOP and the server finish exchanging sensitive data, the server can actively terminate the connection, and resume the session between itself and the browser.

D. Deployment Modifications

Both a client and a server's OS needs to be modified to support SwitchMan's new TCP options and SwitchMan functions. In addition, we also need to modify a web server to deploy SwitchMan. The server software must separate sensitive data from non-sensitive data. But SwitchMan does not require client applications to be modified.

V. Implementation

As a proof of concept, we implemented SwitchMan using the open source platform Linux. We implemented the `SwitchMan` component as a daemon process running with root privilege. It starts a user's account on two different virtual terminals: the default `/dev/tty7` for a user's regular account, and another available terminal `/dev/ttyn` for a user's protected account, where `n` is a number ranging from 1 to 6. The `SwitchMan` component is in charge of three tasks: 1) extracting a server's TLS certificate; 2) invoking a user's TIOP program; and 3) switching a user to the virtual terminal running under his protected account. Our current implementation has 4900 lines of C code. We implemented the TIOP program using the X11's GUI.

We implemented the `Filter` component and the new TCP option field as a kernel patch to Linux 3.13.11. We modified the kernel's TCP handling code to add and strip off the TCP options. When a server requests a trusted I/O path, the `Filter` component strips off $nonce_1$ and $session\_id$ fields carried by a TCP option, and passes them to `SwitchMan`, which in turn passes them to a user's TIOP program. The total patch is around 200 lines of C code.

A SwitchMan-enabled server uses the same kernel to add and extract new TCP options. We implemented a server application which generates the switching request, and splits content into normal and sensitive data. The total changes is around 900 lines of Java code.

VI. Evaluation

In this section, we evaluate the design and implementation of SwitchMan from three aspects: usability, security and performance.

1) Usability. A strong motivation of this work is to make SwitchMan easy to use. We compare its usability with related work.
2) Security. Due to space limit, we omit a thorough security analysis of the SwitchMan design under various threats. Instead, we use the size of TCB of a solution as the security indicator.
3) Performance. Finally, we evaluate SwitchMan's performance overhead using a prototype implementation. We measure both the client and server's communication overhead.

We compare SwitchMan with three other systems to evaluate its strengths and weaknesses: Qubes OS [9], Cloud Terminal [21], and BitE [3]. Qubes OS is a secure desktop operating system that isolates different applications into different virtual machines. Cloud Terminal installs a secure thin terminal on a user's computer and moves all other application logics to a cloud. Each application runs inside a separate VM on a cloud and a user only uses the secure thin terminal for input and output. BitE uses a mobile phone as a secure input/output device.

A. Usability

Evaluating the usability of a security system is challenging, as there is no single usability metric that measures the usability of a system. To address this challenge, we use several usability factors to analyze SwitchMan's user friendliness.

Nothing-to-carry considers whether a user needs an additional physical device (e.g., a phone) to use a system.

No user management effort means a user does not need to manually manage the switching between a trusted I/O path and an untrusted one.

No noticeable performance degradation means a user does not experience noticeable slow down when using a system.

Comparison: Table I shows the comparison results. As

Table I
THIS TABLE COMPARES SWITCHMAN WITH THREE OTHER SYSTEMS FROM MULTIPLE DIMENSIONS.

factor	Qubes OS	Cloud Terminal	BitE	SwitchMan
USABILITY				
Nothing-to-carry	✓	✓	✗	✓
No user management effort	✗	✗	✗	✓
No noticeable performance degradation	✗	✓	✓	✓
SECURITY				
TCB size	VMM + guest OS kernel + graphic system	kernel modules + hypervisor + cloud	kernel + mobile OS	kernel + graphic system

can be seen, SwitchMan is one of the most user friendly system. It does not require a user to carry any additional device and requires no user management effort in switching between the trusted and untrusted I/O paths. As we soon show, SwitchMan introduces a low latency when a user switches to his protected account. BitE requires a user to carry a mobile device. Qubes requires a user to manage the switching between different VMs, while Cloud Terminal requires a user to use a special escape sequence to launch the terminal. In addition, Qubes OS runs one VM for applications in one domain, imposing significant memory and computational overhead. Furthermore, different from SwitchMan, Cloud Terminal, Qubes OS, and BitE all require a user to determine the sensitiveness of data, and initiate the I/O path switching. If a user fails to identify the sensitiveness of data or forgets to initiate the switching, these solutions cannot help. In contrast, SwitchMan does not require user management, and lets a server initiate the switching from an untrusted I/O path to a trusted one.

B. *Security*

We use the size of TCB to estimate the security level of a system. As can be seen in Table I, SwitchMan's TCB size is on par with that of Qubes OS and that of BitE, but is larger than Cloud Terminal's. Qubes OS uses a VM to isolate the guest OS each application runs on, but Qubes itself, the administrative VM, and the guest OS that runs the trusted applications must be trusted. Cloud Terminal uses the network protocol stack provided by a commodity OS and it needs to trust this part of the OS and a set of binaries. BitE trusts the OS kernel. In addition, BitE also trusts the additional mobile devices. Nowadays the size of a mobile phone OS is similar to that of a PC OS. So we consider the size of BitE and that of SwitchMan are similar.

C. *Performance*

For all the following experiments, we use two Dell Optiplex PCs with Intel Core i7 CPU 860 @ 2.80GHz and 8 GB RAM as the client and the server machine. Both machines use Ubuntu 18.04.

I/O Path Switching Latency: We test the I/O path switching latency using our prototype implementation of SwitchMan. In our experiments, we access a test web page from a user's regular account, and switch a user to his protected account when receiving a server's switching request. We repeat the experiments $10k$ times and measure the average latency. The experiments show that the average latency for switching is around $14ms$.

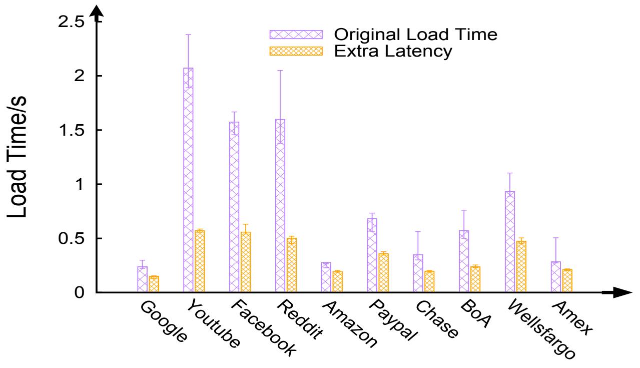

Figure 4. **This figure shows the original page load time of the top five Alexa [27] websites and top five bank websites, and the extra latency added by SwitchMan.**

Extra Latency: We run experiments to measure the extra latency SwitchMan introduces to access a normal web page. The latency is caused by the HTTPS/TLS connection established by TIOP. We connect a client to one of the top five websites and top five bank sites ranked by Alexa [27]. We assume each of the web pages has a login button. We then simulate the case where a user clicks on the login button and the server requests a trusted I/O path for the user to input his login information. We measure the time from the client's OS receiving the server's request to TIOP finishing loading the login page. We consider this time as the extra latency to load the entire page. We repeat the experiments for each site 500 times, and measure the average extra latency and standard deviation, and show them in Figure 4. As can be seen, the extra latency is less than 0.5s for all sites. We consider this extra latency acceptable for improved security.

Client Computational Resource Overhead: We measure the extra computational resource overhead introduced by SwitchMan to a client computer. With SwitchMan, a client's computer needs to run an extra graphic server, the kernel

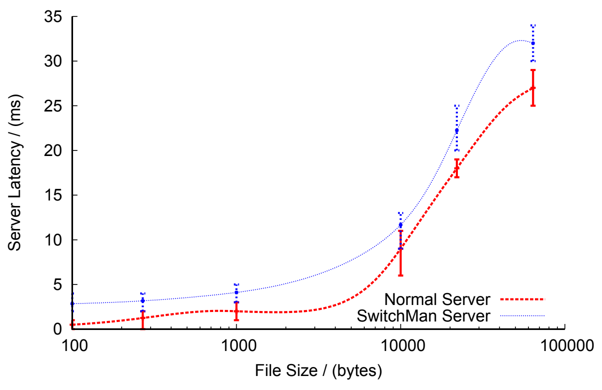

Figure 5. **The latency at the server side when sending files with different sizes.**

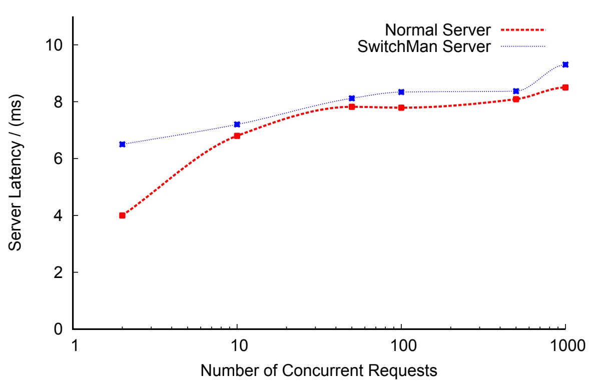

Figure 6. **The latency at the server side when there are concurrent requests.**

`SwitchMan` module, and TIOP. We measure how much these processes cost. For our implementation, the total memory cost of all SwitchMan's components is $280M$. And there is nearly no difference in the number of system operations per second and CPU resources consumption, since the extra graphical server is running at the background and usually stays in a "sleep" condition. We also measure the kernel module `SwitchMan`'s memory consumption: when `SwitchMan` is running as a daemon, it takes $15160k$ mapped memory. We consider this overhead acceptable, given that modern PCs typically have at least a few gigabytes of memory.

Memory Cost Comparison: The memory cost of our experimental machine during idle time is 0.933GB. When SwitchMan is turned on, the memory cost is 1.213GB. When we run Qubes OS with 5 VMs, the cost is 2.870GB. This is because SwitchMan just uses an extra virtual terminal to launch another graphical server, while Qubes uses different VMs to launch an operating system module.

D. Server Evaluation

Server Side Latency: We test the overhead on the server side when a server sends files with varying sizes. The server side overhead mainly comes from the additional SNP data a sever sends at the connection setup time and during the client account switching time. We measure the time from when a connection is established to when it is finished.

Figure 5 shows that the server side latency is relatively fixed and does not vary by file size. This is because the additional overhead is only added at Step 1 and 2 of the SNP protocol, and does not affect the actual data transmission. The additional latency is less than 10ms for all file sizes.

Concurrency: We let the client machine send concurrent requests to the server to evaluate how the server performs when there are multiple simultaneous connections. For this experiment, we use a file size of 300 bytes. Figure 6 shows the results. With the increasing number of concurrent requests, the average page load time slightly increases for both SwitchMan server and a normal server. SwitchMan adds a small latency compared to the normal server when the number of concurrent requests increases from two to 1000.

VII. Discussion and Future Work

In this section, we discuss a few additional design issues and future work.

SwitchMan needs OS modifications but does not require modifications of client applications. Although we have kept the modifications minimal and much fewer than previous proposals [3], [9], [21], we acknowledge that such modifications may not be adopted by OS vendors. However, our goal as researchers is to provide an easy to use security alternative for the real world to choose from.

We use a server initiated switching design. This design may enable a malicious server to unnecessarily switch a user to his protected account, launching a denial of service attack. Malicious client software cannot launch such an attack. SwitchMan's design allows a user to use a special keystroke sequence to leave the protected account, and future work can add a blacklist option to allow a user to blacklist such a malicious server.

One might argue that it is a client's interest to protect its sensitive data. Therefore a server-initiated protection mechanism has ill-aligned incentives. However, in practice, since there are multiple services competing for customers, we believe they have incentives to offer security-enhanced services to their customers. For instance, a bank may desire to adopt our solution to prevent its customers' passwords from being stolen, and customers may prefer a bank that offers such a solution.

Finally, we currently design TIOP as a simple HTML browser without advanced features such as Javascript to avoid security vulnerabilities. Future work can extend TIOP to be a secure and fully functioning browser.

VIII. Conclusion

In this paper, we present SwitchMan, an architecture that enables a server to automatically switch a user to a secure terminal for sensitive user input/output. Our experiments and analysis suggest that SwitchMan is lightweight and easy to use. Protecting a user's sensitive information from being stolen by malware remains an open problem. We believe SwitchMan offers a valuable design alternative for the real-world to adopt.

Acknowledgment

We thank our shepherd Yod-Samuel Martín and the anonymous reviewers for their valuable suggestions.

References

[1] Kaspersky Lab, " The Great Bank Robbery: Carbanak APT," https://securelist.com/the-great-bank-robbery-the-carbanak-apt/68732/, 2015.

[2] Intel, "Intel Identity Protection Technology with Protected Transaction Display," https://www.intel.com/content/www/us/en/architecture-and-technology/identity-protection/identity-protection-technology-general.html, 2012.

[3] J. M. McCune, A. Perrig, and M. K. Reiter, "Bump in the ether: A framework for securing sensitive user input," in *USENIX ATC*, 2006.

[4] J. M. M. A. Perrig and M. K. Reiter, "Safe passage for passwords and other sensitive data," in *NDSS*, 2009.

[5] M. Mannan and P. C. Van Oorschot, "Using a personal device to strengthen password authentication from an untrusted computer," in *International Conference on Financial Cryptography and Data Security*, 2007.

[6] R. Sharp, A. Madhavapeddy, R. Want, and T. Pering, "Enhancing web browsing security on public terminals using mobile composition," in *ACM MobiSys*, 2008.

[7] M. Wu, S. Garfinkel, and R. Miller, "Secure web authentication with mobile phones," in *DIMACS workshop on usable privacy and security software*, 2004.

[8] C. Yue and H. Wang, "Sessionmagnifier: A simple approach to secure and convenient kiosk browsing," in *ACM UbiComp*, 2009.

[9] J. Rutkowska and R. Wojtczuk, "Qubes OS Architecture," *Invisible Things Lab Tech Report*, p. 54, 2010.

[10] B. Lampson, "Accountability and Freedom," in *Cambridge Computer Seminar, Cambridge, UK*, 2005, pp. 1–26.

[11] M. James, "Secure and Simple Sandboxing in SELinux," https://www.slideshare.net/jamesmorris/secure-and-simple-sandboxing-in-selinux, 2009.

[12] E. Walsh, "Application of the flask architecture to the x window system server," in *Proceedings of the 2007 SELinux Symposium*, 2007.

[13] N. Feske and C. Helmuth, "A nitpickers guide to a minimal-complexity secure gui," in *IEEE ACSAC*, 2005.

[14] Y. Cheng, X. Ding, and R. H. Deng, "Driverguard: Virtualization-based fine-grained protection on i/o flows," *ACM Transactions on Information and System Security*, vol. 16, no. 2, 2013.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SOSP*, 2003.

[16] P. C. Kwan and G. Durfee, "Practical uses of virtual machines for protection of sensitive user data," in *Springer ISPEC*, 2007.

[17] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," in *IEEE Security & Privacy*, 2009.

[18] H.-M. Sun, Y.-H. Chen, and Y.-H. Lin, "opass: A user authentication protocol resistant to password stealing and password reuse attacks," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 2, 2012.

[19] IBM Zurich Research Lab, "IBM Zone Trusted Information Channel," http://www2.sta.uwi.edu/anikov/info1400/lectures/08-ITF-video-case-1-IBM-Zone-Trusted-Information-Channel-(ZTIC).pdf, 2008.

[20] K. Borders and A. Prakash, "Securing network input via a trusted input proxy." in *USENIX HotSec*, 2007.

[21] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, "Cloud terminal: secure access to sensitive applications from untrusted systems," in *USENIX ATC*, 2012.

[22] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *IEEE Security & Privacy*, 2012.

[23] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0." in *USENIX Security Symposium*, 1999.

[24] Sourceforge, "Integrity Measurement Architecture (IMA)," https://sourceforge.net/p/linux-ima/wiki/Home/.

[25] CVE, "Common vulnerabilities and exposures," https://cve.mitre.org.

[26] M. Blaze, "A cryptographic file system for unix," in *ACM CCS*, 1993.

[27] Alexa, "The top 500 sites on the web," https://www.alexa.com/topsites, 2017.