BUILDING TRUST IN THE USER I/O IN COMPUTER SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Yeongjin Jang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2017

BUILDING TRUST IN THE USER I/O IN COMPUTER SYSTEMS

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Taesoo Kim, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Professor Kang Li
Department of Computer Science
University of Georgia

Professor Yongdae Kim
Department of Electrical Engineering
KAIST

Date Approved: 24 July 2017

To my dear wife,

Sejin Keem,

and my parents,

Bongsik Jang and Sook-Kyeong Cho,

for all the love and support.

ACKNOWLEDGEMENTS

First of all, I am deeply grateful to my advisor Professor Wenke Lee for his guidance and support through the Ph.D. program. Research discussions that he and I had for around seven years enlighted me a lot, and the experience that I gained through these conversations raised me as a decent researcher who will now start contributing back to the society. My co-advisor Professor Taesoo Kim was a great source of research insight and stimulating me a lot with his technical expertise. He was not only a brilliant mentor but also a great friend.

I would also like to acknowledge my thesis committee members: Professor Mustaque Ahamad, Professor Kang Li, and Professor Yongdae Kim, for willing to serve on my dissertation committee. Their insightful comments and suggestions have enlightened me to make significant improvements to this thesis. I would like to acknowledge Professor Kang Li, for his help in keeping my focus on both sides of academic research and practice by working together in the "Disekt" capture-the-flag (CTF) team. Thanks to Professor Yongdae Kim, for being a sagacious mentor and who greatly inspired me to pursue the academic career.

Many students and researchers in GTISC collaborated with me on this work including Simon Chung, Tielei Wang, Billy Lau, Sangho Lee, Bryan Payne, Paul Royal, Long Lu, Chengyu Song, Byoungyoung Lee, Xinyu Xing, Yizheng Chen, Wei Meng, and Kangjie Lu, and even more students and researchers in System Software and Security Lab (SSLab) including Changwoo Min, Insu Yun, Meng Xu, Wen Xu, Ren Ding, and Jinho Jung were always happy to discuss research challenges and helped me a lot in the day to day battle of a graduate student's life. I also thank my research collaborators at KAIST, including Dongkwan Kim, Hongil Kim, Jaehyuk Lee, and Professor Brent Byunghoon Kang.

In addition to my collaborators, I would especially like to express my sincere gratitude

and great appreciation to the Kwanjeong Educational Foundation for their generous support of my Ph.D. study, which kept me free from all financial issues.

Alongside the research, my friends Sehoon Ha, Dong-Gu Choi, Hanju Oh, Philip Kwon, Gee Hoon Hong, Hwajung Hong, and Yusun Lim, and the members of the Post Collier Hills Tennis Club, Mincheol Chang, Kwangsup Eom, Jaehan Jung, Seongjun Kim, Kwangho Park, Chanyeop Park, and Pyungwoo Yeon, they all made my life in Atlanta bright even when I was intimidated by research related stress.

Last but not least, I would like to give special thanks to my family for their continuous love and support. My dear wife, Sejin Keem, she always supported and encouraged me while she was also going through a tough journey for her Ph.D. study. Thank you, Dr. Keem. My parents Bongsik Jang and Sook-kyeong Cho, they always sent their loving heart with full of warmth to me so I can keep my smiling face even when my paper got harshly rejected. I also send my thanks to my father-in-law, Changho Keem and my mother-in-law, Kyunghee Choi, for their support and encouragement at the toughest moment of my Ph.D. study.

Although my seven-year journey for finishing this dissertation was not always filled with all-happy-moments as all life does, I can sustain myself and complete this journey because of your support. I always carry all of you in my heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

User input plays an essential role in computer security because it can control system behavior and make security decisions in the system. System output to users, or user output, is also important because it often contains security-critical information that must be protected regarding its integrity and confidentiality, such as passwords and user's private data. Despite the importance of user input and output (I/O), modern computer systems often fail to provide necessary security guarantees on them, which could result in serious security breaches.

This dissertation aims to build trust in the user I/O in computer systems to keep the systems secure from attacks on the user I/O. To this end, we analyze the user I/O paths on popular platforms including desktop operating systems, mobile operating systems, and trusted execution environments such as Intel SGX, and identified that threats and attacks on the user I/O can be blocked by guaranteeing three key security properties of user I/O: integrity, confidentiality, and authenticity.

First, **GYRUS** addresses the integrity of user input by matching the user's original input with the content of outgoing network traffic to authorize user-intended network transactions. Second, **M-AEGIS** addresses the confidentiality of user I/O by implementing an encryption layer on top of user interface layer that provides user-to-user encryption. Third, the **A11Y ATTACK** addresses the importance of verifying user I/O authenticity by demonstrating twelve new attacks, all of which stem from missing proper security checks that verify input sources and output destinations on alternative user I/O paths in operating systems. Finally, to establish trust in the user I/O in a commodity computer system, I built a system called **SGX-USB**, which combines all three security properties to ensure the assurance of user I/O. **SGX-USB** establishes a trusted communication channel between the USB controller and an enclave instance of Intel SGX. The implemented system supports

common user input devices such as a keyboard and a mouse over the trusted channel, which guarantees the assurance of user input.

Having assurance in user I/O allows the computer system to securely handle commands and data from the user by eliminating attack pathways to a system's I/O paths.

CHAPTER I

INTRODUCTION

1.1 Motivations and Goals

User input plays an essential role in computer security because it can control system behavior and make security decisions in the system. System output to users, or user output, is also important because it often contains security-critical information that must be protected regarding its integrity and confidentiality, such as passwords and user's private data. Despite the importance of user input and output (I/O), modern computer systems often fail to provide necessary security guarantees on them, which could result in serious security breaches. Hence the machine infected by malware can send/alter the command that user has never supplied, and keyloggers can steal passwords from the system; cloud service providers can easily read user's messages, and any attackers can take control of the device across the security domain by injecting a fake user input to a system.

To block a class of attacks that target weaknesses in the user I/O in computer systems, this dissertation aims to build trust in the system's I/O path to keep the systems secure by fundamentally cutting off attack pathways. To this end, we analyzed the user I/O paths on popular platforms including desktop operating systems, mobile operating systems, and trusted execution environments such as Intel SGX, and identified that threats and attacks on the user I/O can be blocked by guaranteeing three key security properties of user I/O: integrity, confidentiality, and authenticity.

This dissertation includes *four* projects that address the *three* essential security properties of user I/O: integrity (**GYRUS** [85] in §2), confidentiality (**M-AEGIS** [101] in §3), and authenticity (**A11Y ATTACK** [86] in §4), and the **SGX-USB** project (in §5) guarantees assurance as a combination of the three key security properties in order to enable secure user I/O to Intel

SGX.

1.2 Dissertation Overview

1.2.1 The Integrity of User Input

GYRUS [85] (in Chapter §2) is a security system that protects the integrity of text user input that can be sent as security sensitive network transactions such as sending an e-mail message and online banking transactions by allowing only user-intended network transaction to be sent from a system even if the operating system is compromised by attacks. GYRUS takes a new approach on inferring the user intent from input because prior attempts on extracting user intent from keystrokes is a difficult problem by cause of the complexity of text operations. The key observation on keystroke user interaction is that the user not only uses the input device but also looks at the screen output to check if his/her typing is captured correctly. GYRUS exploits this feedback loop to capture the text data and the user's intent from the UI layout, such as text boxes on the screen. Then, GYRUS enforces the policy called "*What You See Is What You Send (WYSIWYS)*" to match the user's intent with the outgoing network traffic. Under this policy, GYRUS preserves the integrity of text that embodies user intent, and allows the network transaction only if the network traffic matches the prior intent as shown as the text on the UI. The enforcement of this policy either requires a direct matching of text or allows a simple transform (e.g., encoding) from the captured text from UI to the content in the network traffic, which many text-based applications meet this requirement. GYRUS can be applied to protect several real-world applications, including e-mail messages in Outlook, postings on Facebook, and the financial transactions in online banking applications such as Paypal. Evaluation of GYRUS against real-world malware has shown that GYRUS blocks all malware-generated network traffic while allowing all the user intended transactions, and incurs negligible performance overhead (only 39 ms of delay on each keystroke) while adding less than 6% of network latency.

1.2.2 The Confidentiality of User I/O

Public messaging services, such as GMail, Outlook, Facebook Messenger, etc., claim to provide the end-to-end encryption by using Transport Layer Security (TLS/SSL). Unfortunately, the encryption is only from the user to the server, not from the user all the way through to the other users, which means that the server can access the plaintext data. Since applying a user-to-user encryption like PGP is difficult because of the change in user experience and requirement of protocol reverse-engineering, **M-AEGIS** [101] (in Chapter §3) took a new approach. M-AEGIS is a system that not only provides the user-to-user confidentiality of plaintext data but also preserves the user experience through the creation of a UI overlay called *Layer 7.5*, which is interposed between the application (OSI Layer 7) and the user (Layer 8). This approach allows M-AEGIS to implement a true user-to-user encryption of data while achieving goals in security, usability, and adaptability. To preserve the exact application workflow and look-and-feel, M-AEGIS uses Layer 7.5 to put a transparent window on top of existing application GUIs to both intercept plaintext user input then encrypt before feeding it to the underlying app, and to decrypt the (encrypted) data from the app before displaying the (plaintext) data to the user. Moreover, to support the search operations, which is essential in e-mail services, M-AEGIS implements the *easily-deployable efficiently-searchable symmetric encryption scheme* (EDESE) to enable text search operation over the encrypted text data. This technique allows M-AEGIS to be transparently integrated with the most of messaging services without hindering usability or requiring reverse engineering. A prototype of M-AEGIS is implemented on Android and demonstrates that it can support a number of popular services, e.g., Gmail, Facebook Messenger, WhatsApp, etc. The performance evaluation and the user study show that M-AEGIS incur minimal overhead and no compatibility problem when adopted on Android.

1.2.3 The Authenticity of User I/O

The **A11Y ATTACK** [86] (in Chapter §4) is a new class of attacks caused by the failure to properly verify (i.e. authenticate) the source and the destination of the user I/O on operating systems. Major operating systems (i.e., Windows, Linux, Android and iOS) provide the means of programmatically generating user input and programmatically reading UI output to build assistive technology for supporting accessibility (a11y) for disabled users, in which the federal law mandates the support. Unfortunately, having such alternative ways of generating inputs and reading outputs introduces new security vulnerabilities. For example, an audio speech generated by an unprivileged application can control privileged applications through the voice commander on the system. The A11Y ATTACK defines assistive technologies as I/O subsystems that either transform user input into interaction requests for other applications and the underlying OS, or transform application and OS output for display on alternative devices. While the user I/O is considered critical for the system's security, inadequate security checks on these new I/O paths make it possible to launch attacks from accessibility interfaces. The A11Y ATTACK evaluated accessibility supports for four popular operating systems: Microsoft Windows, Ubuntu Linux, iOS, and Android. We identified *twelve* new attacks that can bypass state-of-the-art defense mechanisms deployed on these OSes, including mandatory access control in both Windows and Linux, and bypassing sandbox in both iOS and Android. Further analysis illustrates that the root cause of the attack is that the design and implementation of accessibility support involve inevitable trade-offs among compatibility, usability, security, and (economic) cost. The A11Y ATTACK also proposed a number of countermeasures to either make the implementation of all necessary security checks easier and intuitive, or to alleviate the impact of missing/incorrect checks.

1.2.4 The Assurance of User I/O

One promising approach to protecting a system from software based attacks is to implement the anchor of security in hardware. Processor chips from major manufacturers, such as

Intel, have recently embedded several new hardware extensions for the trusted execution environment (TEE) such as Intel Software Guard Intension (SGX). Although the trsuted execution environment is available in hardware today, unfortunately, the effect of them does not come out as expected because Intel SGX cannot support user-facing applications due to missing trusted user I/O path. In order to harvest the security benefits from the TEE, We design **SGX-USB** (in Chapter §5), which establishes a trusted I/O path between a USB port and the trusted execution environment (TEE). The design of **SGX-USB** introduces a new trusted hardware, the USB Proxy Device at a USB port of a system, and this hardware component establishes a trusted communication channel between the USB port and an enclave instance through the remote attestation process of Intel SGX and its extension. After establishing a trusted channel, **SGX-USB** can support typical user input devices such as keyboard and mouse as well as more complex user-facing devices such as camera, microphone, speaker, and display devices by forwarding USB packets over the trusted channel. Because the trusted channel guarantees all three security properties, such as integrity, confidentiality, and authenticity, **SGX-USB** can ensure the assurance of user input and allows the enclave instance to handle commands and data from the user securely. While the currently discussed applications of Intel SGX only perform the network and the file I/O securely, this new design enables secure user I/O in the TEE so that Intel SGX can facilitate user-facing trusted applications, such as authentication manager and end-to-end encrypted video chat application.

CHAPTER II

GYRUS: A FRAMEWORK FOR USER-INTENT MONITORING OF TEXT-BASED NETWORKED APPLICATIONS

2.1 Motivation

Host-based security systems have traditionally focused on detecting attacks. Misuse detection targets attacks that follow predefined malicious patterns, whereas anomaly detection identifies attacks as anything that *cannot* be the result of correct executions under any input or execution environment. Over the time, systems following this approach have shown that they usually have a too narrow definition of "attacks", which often necessary to keep their false positive rate acceptable. Thus misuse detection generally cannot detect new attacks, while anomaly detection is known to suffer from mimicry attacks.

Instead of perpetuating the cycle of attack analysis, signature creation, and blacklist updating, we believe a more viable approach is to create an accurate model of what is the correct, user-intended behavior of an application, and then ensure the application behaves accordingly. The idea of defining correct behaviors of an application by capturing user intent is not entirely new, but previous attempts in this space use an overly simplistic model of the user's behavior. For example, they might infer a user's intent based on a single mouse click without capturing any associated context. While in some cases (e.g. ACG [133]), the click captures all the semantics of the user's intent (e.g. access the camera), in other cases (e.g. BINDER [38] and Not-a-Bot [72]), the user's intent involves a richer context, and failure to capture the full semantics will again allow for attacks to disguise as a benign behavior. For example, imagine a user who intends to send $2 to a friend through PayPal. A mouse click can identify the user's intent to transfer money, but not the value or recipient of the transfer. So this $2 transfer to a friend could become a $2,000 transfer to an unknown

person. Without context, it is simply impossible to properly verify a user's intent, regardless of if we are protecting a financial transfer, an industrial control system, or a wide range of other user-driven applications.

In this chapter, we propose a way to capture the richer semantics of the user's intent. Our method is based on the observation that for most text-based applications, the user's intent will be displayed entirely on screen, as text, and the user will make modifications if what is on screen is not what she wants. Based on this idea, we have implemented a prototype called **GYRUS**[1], which enforces correct behavior of applications by capturing user intent. In other words, **GYRUS** implements a "What You See Is What You Send" (WYSIWYS) policy. **GYRUS** assumes a standard VM environment (where **GYRUS** lives in the dom-0 and the monitored applications live in dom-U[2]). Similar to BINDER and Not-a-Bot, **GYRUS** relies on the hypervisor to capture mouse clicks from the user, and use these as an indication that the user intends the application to perform certain actions. To capture the semantics of user intent that cannot be inferred from just observing a mouse click, we take the approach of drawing what we think the user should see in the dom-0. In particular, the dom-0 will draw a secure overlay on top of the dom-U display window (the VNC viewer in KVM environment), covering editable text area of targeted applications in dom-U, while leaving the rest of the dom-U display visible. *We stress that this rendering is isolated from dom-U – software in dom-U cannot overwrite or modify what has been drawn.* Since we render all editable text the user sees, we can easily confirm that what is intended is what we have drawn. *By drawing all the text the user is supposed to see in our overlay,* **GYRUS** *can also handle scrolling properly.* Even if only part of the text is displayed at any time, **GYRUS** can keep track of what has been displayed over time and derive the full content of the user intended input.

To determine what text to display in the overlay, we deploy a component called the *UI*

[1] The fusiform gyrus is a part of the human brain that performs face and body recognition.

[2] In this chapter, we adopt the terminology from the Xen community. In other settings, the dom-0 is referred to the Security-VM, while dom-U is referred to the Guest-VM.

monitor in dom-U. We stress that the UI monitor is not trusted since incorrect behavior in this component will be immediately noticed by the user and only result in a denial-of-service (DoS) in the worst case. The UI monitor is also responsible for telling the dom-0 logic the location of buttons that signify the user's intent to commit what is displayed to the network (e.g. the "send" button in an email client), and when the user finally clicks on such buttons. Using such information, **GYRUS** will make sure the outgoing network traffic matches the text displayed. In short, **GYRUS** enforces the integrity of user-generated network traffic and prevents malware from misusing network applications to send malicious traffic even if the malware mimics legitimate applications by running an application's protocol correctly or injects itself into benign applications. Note that **GYRUS** only checks network traffic under protocols used by the protected applications, and it does not interfere with traffic from other applications, such as background services, RSS feed readers, and BitTorrent clients. Additionally, **GYRUS** can support asynchronous or scheduled traffic like e-mail queued for sending in the future. From our evaluation, **GYRUS** exhibits good performance and usability, while blocking all tested attacks.

Any attempt to make sure an application behaves according to user intent will have some application-specific logic, and **GYRUS** is no exception. This is inherently true for our approach because: 1) different applications will have a different user interface, and thus user intent will be interpreted differently and, 2) different applications will have different logic for turning user input into network traffic or other forms of output. The best we can do is to make the per-application logic as easy to build as possible. In **GYRUS**, we simplify the UI-related part of the per-application logic by making use of an existing library called UI Automation, which is for supporting assistive technologies and UI testing. As for the logic to map user intent to the expected behavior of an application, the complexity mostly depends on the application, and **GYRUS** and the WYSIWYS policy is not suitable for all applications. In particular, applications with an arbitrarily complex encoding of their text, or those using proprietary protocols cannot be easily supported by **GYRUS**. Nevertheless,

we have shown that it can be used on email clients, instant messenger applications, online social network services and even online financial services. §2.5 discusses what applications are best protected by **GYRUS**.

The per-application development cost for **GYRUS** is justifiable since **GYRUS** is attack-agnostic: **GYRUS** makes assumptions about *what* the attackers are trying to achieve but not *how*. In other words, once one builds the logic for an application, **GYRUS** will be able to protect that application against an entire class of attacks, no matter how attacks evolve. Therefore, over time, the cost of deploying **GYRUS** will be lower than existing host-based security systems, which usually need continuous updating to stay current with the latest attacks.

Finally, we emphasize that **GYRUS** does not replace existing host-based security systems. Instead, **GYRUS** uses a different philosophy to fill a gap in traditional security systems by defining and monitoring *normal* behavior. Thus, **GYRUS** fits best when it is used to complement other security systems, such as antivirus, firewalls, and intrusion detection systems (IDS).

The primary contributions of this chapter include: 1) the "What You See Is What You Send" concept that includes securely capturing what the user sees on the screen at the time an event triggers outgoing traffic. Using this, we can determine what the user intended outgoing traffic should be for an important class of applications. Furthermore, our idea is transparent to the OS and applications, and only requires standard assumptions about the virtualized environment. 2) The demonstration of how we can use common features such as accessibility libraries[3] for inter-VM monitoring without knowing the internals of the monitored applications. And 3) the demonstration of the viability of **GYRUS** by implementing the framework along with support for real-world applications in Microsoft Windows 7. The prototype of **GYRUS** currently supports email, instant messaging, social

[3] Similar capabilities should be available on most systems that support screen reader for visually impaired users.

networking applications, and online financial applications, effectively covering the most common network applications in everyday use.

2.2 Related Work

This section discusses the related work and how **GYRUS** improves on the current state-of-the-art. The discussion is also intended to provide some context for our work. We group the related work into three areas: 1) capturing user intent, 2) trusted execution environment, and 3) verifiable computation.

Capturing Human Intent. Like **GYRUS**, BINDER [38] and Not-A-Bot (NAB) [72] also try to determine if outgoing traffic is legitimate based on observed human intent; in particular, both systems enforce a policy which states that *outbound network connections that come shortly after the user input are user intended*. However, as mentioned in §2.1, in some cases only capturing the timing of user-generated events is not enough. In contrast to BINDER and Not-A-Bot, **GYRUS** captures more semantics of the user's intent, so only traffic with the correct content can leave the host. Additionally, since BINDER and Not-A-Bot use timing information to determine if traffic is user intended, they cannot handle asynchronous network transactions (such as emails queued to be sent later). **GYRUS** solves this problem by relying on the semantics, but not the timing of user generated events, and by decoupling the capturing of user intent from the enforcement of its traffic filtering policy.

User-Driven Access Control [133] captures the user's intent for security purposes using an access control model that grants permissions based on a user's GUI interactions. It uses access control gadgets (ACGs) to capture a user's intent. Clicking on an ACG grants permission on a resource associated with the ACG. **GYRUS** uses a similar approach on UI widgets to identify traffic-triggering user input. However, in User-Driven Access Control, the permission is bound to certain user-owned resources, not to the *content* the user intends to send to these resources. In other words, when the user clicks on an ACG that has permission to use the network device, any outgoing traffic, even with malicious intent, will be allowed.

On the contrary, GYRUS captures both the user's intent to send something and also the intended content of that outgoing traffic and can stop any unintended network traffic.

Trusted Execution Environment. Virtualization has enjoyed a resurgence in popularity in recent years. Proponents have argued that by using small, verifiable hypervisor kernels, the isolation of one virtual machine from another can be assured [108, 76]. Recent research has aimed to enhance this security by reducing the size of the hypervisor's code [144, 162], modularizing its components [35], or verifying its security [96]. These isolation properties make virtualized environments an attractive way to implement security applications. Virtualization-based solutions have been used to implement trusted computing architectures [57, 111], intrusion detection systems [58], malware analysis systems [89], and zero-day intrusion analysis systems [91]. However, none of these take user intent into account. Thus, we believe GYRUS can enrich research in each of these areas by showing how to build on the isolation provided by a virtualized environment to perform simple checks that will improve the system's security.

Verifiable Computation. GYRUS has some common goals with the field of verifiable computation, which has focused on ensuring correct code execution by an untrusted third party. This work has taken many forms including general-proof protocols [66, 67, 59], Probabilistically Checkable Proofs (PCPs) [12, 138, 139], or relying on fully-homomorphic encryption (FHE) [60, 33]. While these systems can prove that a third party has processed a requested execution correctly, they cannot tell whether the input of this execution is correct. GYRUS fills this gap by checking that the input used for computation is what was provided by the system's user. GYRUS then completes the validation by also checking whether the outcome (e.g, network packet) of application execution is the correct result for a given input. Recent work [128] shows that verifiable computation can be used in practical settings, so we believe that the complementary aspects of GYRUS and verifiable computation could prove to be a powerful combination in future security systems.

2.3 Overview

In this section, we present a high-level overview of GYRUS. First, we describe our threat model, and then we introduce a policy called "What You See Is What You Send" (WYSIWYS), which is integrated and enforced by GYRUS to address the threat model. Then, we describe the essential elements of GYRUS and discuss suitable applications of GYRUS.

2.3.1 Threat Model

GYRUS is designed to utilize a standard virtualized environment with a hypervisor (VMM), a trusted dom-0 that executes most parts of GYRUS, and an untrusted dom-U that runs the applications to be protected as well as with some untrusted components of GYRUS. We collect data for determining a user's intent from the hardware input and output devices, including the keyboard, mouse, and screen. We make the following security assumptions:

- The hypervisor and dom-0 are fully trusted.
- Attackers cannot have physical access to the machine, and we trust the hardware.
- All hardware input events must be interposed by the hypervisor, and they must first be delivered to dom-0. The hypervisor provides complete isolation of input hardware, preventing hardware emulation originating from dom-U.
- Dom-U is not trusted; therefore it can be compromised entirely.

We stress that we do not apply any security assumption on dom-U. This implies that GYRUS could function correctly even if the dom-U is entirely compromised (including kernel-level attacks). In other words, even though GYRUS extracts information from the memory of dom-U by running a helper component called *UI Monitor* inside of it, we do NOT assume the correctness of such information. Instead, we designed a trusted component called Secure Overlay to verify the validity of this information. Detailed information for these components will be described in the next section.

2.3.2 User Intent

As mentioned in §2.1, the goal of **GYRUS** is to capture rich semantics to understand a user's intent. This semantics is used to ensure that only user intended traffic can leave the system. In this context, user intent is limited to what we can infer from the system's input devices. In BINDER and Not-a-Bot, user intent is captured by directly observing input hardware events (mainly from keyboard and mouse). However, this approach is limited due to the missing contextual information and the challenges of reconstructing user content without "seeing" the screen. To make a sound security decision, we must capture further details about the user's intent. For example, the task of reconstructing a message from a mail client using only keystrokes and mouse clicks would require us to reconstruct the entire windowing system and the logic behind text boxes (e.g. how to update the location of the caret upon receiving keyboard/mouse input), as well as to reproduce the logic to handle application-specific function keys.

2.3.3 What You See Is What You Send

Instead of capturing and reconstructing user intent strictly from hardware input events, our solution is to monitor output events from the target applications. The main observation behind our approach is that in almost all text-based applications, the text that the user types will be displayed on the screen. This output allows the user to know that she has typed correctly and made the necessary correction when there is a mistake. Therefore, we can capture an accurate representation of the user intent if **GYRUS** can "see" what a user sees. With this information, we can determine what the user-intended outgoing traffic should look like and make sure that this is the only traffic that the target application sends. We call this approach "What You See Is What You Send" (WYSIWYS).

To enforce WYSIWYS, **GYRUS** is required to correctly and fully capture textual content that is displayed to the user. Additionally, **GYRUS** needs information about the UI structure. In **GYRUS**, we have two components that implement these features: a dom-U component

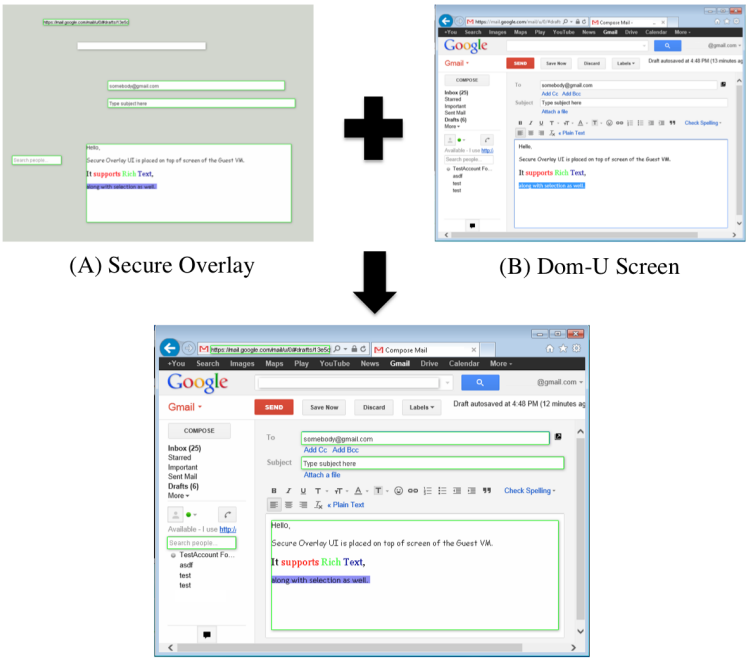

Figure 1: Secure Overlay working with the GMail application in Internet Explorer 10. Overlaid edit controls are highlighted with green bounding boxes. Gyrus changes the border color to red if it detects any infringement.

called *UI Monitor*, which extracts textual content and a high-level UI structure of the current screen, and a dom-0 trusted component called *Secure Overlay* which verifies if the captured text matches the user's intent.

The *UI Monitor* operates on top of the *UI Automation* [120] library in Microsoft Windows, which is originally intended for building accessibility utilities such as screen readers for visually impaired users (i.e., this library is designed to capture text displayed on the screen and fits our purpose very well). Not only does the UI monitor capture the displayed text, but it also allows us to determine if the mouse click event observed by dom-0 signifies the user's intent to commit what is displayed on the screen to the network.

Since the UI monitor relies on the code in dom-U, we stress that we *cannot* and *do not* trust the output of this component. Instead, we use the Secure Overlay to show the data captured by the UI monitor to the user. As a result, the user can either validate what the secure overlay displays by not modifying it, or disagree by correcting what she sees (and

this will be captured by the UI monitor again). We call this idea **reflective verification**.

Figure 1 illustrates how WYSIWYS works with the UI Monitor and the Secure Overlay. The UI Monitor grabs the UI structure information from the current screen, including the location of windows, text boxes, and buttons, along with textual content from the text boxes. Then the Secure Overlay positions a transparent overlay screen, and for each text box on the current dom-U screen, it will dynamically draw a matching text box with the same text content at the exact same location. This Secure Overlay component is always drawn on top of the whole dom-U screen, so it always hides any text boxes of applications running in dom-U. While input interaction stays the same from the user's perspective, the output that user sees is actually the text that is captured by the Secure Overlay. And the text shown on the screen will be updated as the user interacts with the application, so the user will naturally verify that this captured content matches her intent.

GYRUS needs to ensure that for all cases, the text shown on the Secure Overlay is exactly matched with the text that the underlying application is presenting. However, in our reflective verification scenario, the user can only verify changes in the currently visible part of the text. If some lines of text scroll out of view and then get updated while they are hidden, this verification process is no longer valid. To handle hidden updates, **GYRUS** keeps track of the text and its changes. To indicate the status of verification, we place a border around the text box. When everything is as expected, the border is green. When the hidden text changes, the border turns red, indicating that the user needs to manually verify the content. In our experience, **GYRUS** works well with most text boxes for default text typing. Additionally, **GYRUS** can support text-editing features such as cut/copy/paste, automatic spell correction, selection of text from a combo box, etc.

2.3.4 Network Traffic Monitoring

After **GYRUS** captures the user's intent using the UI Monitor and the Secure Overlay, the second part of implementing WYSIWYS is to ensure that the traffic generated by the

Figure 2: An example WYSIWYS applicable operation: adding a comment on a post on Facebook. After typing a message for the comment and pressing the ENTER key, the application generates network traffic which goes to the URL: `https://www.facebook.com/ajax/ufi/add-comment.php`. Between the on-screen text and the outgoing network traffic, there is a direct mapping of user-intended content.

monitored application matches what **GYRUS** expects based on the captured user intent. **GYRUS** assumes that there is a simple mapping between the captured user intent and the outgoing traffic. In other words, the network protocol used in the application must transmit the information displayed to the user directly or with simple modifications (e.g., text represented in XML, or a standard encoding such as Base64 and URL encoding). Even though this assumption does not hold for all applications, we argue that many everyday applications are mostly text-based and have very simple processing to generate outgoing network traffic based on the text input from users. Figure 2 shows an example of a simple mapping between user input and network traffic content.

Finally, note that **GYRUS** only inspects specific types of messages under the protocol used by the protected application(s). **GYRUS** will not interfere with any traffic outside of this scope. Even for traffic originating from target applications, **GYRUS** will only check (and potentially block) traffic that contains user-generated content. For example, for SMTP and

instant messenger protocols, we only check commands for sending messages. For HTTP(S) traffic, we only inspect individual URLs that submit user-intended contents, such as posting Twitter messages, adding comments on Facebook, or sending money on Paypal. In §2.4, we will describe how to identify such traffic using the *User Intent Signature*.

2.3.5 Target Applications

Not all traffic that is observed by **GYRUS** can be traced back to some user action that explicitly expresses her intent to create such traffic. For example, when the user tries to load a web page in the browser, she probably has no knowledge about what further HTTP requests will automatically be generated to download all the images on the loaded pages. In addition to these automatic requests, if the text content of the application is represented using a complex encoding on the network protocol, (e.g., evaluating some functions or encryption), **GYRUS** cannot infer expected output of network traffic. As such, in this work, our focus is on traffic that contains rich semantics about the user's intent, and we consider cases where the user does not have a clear understanding of what traffic their action will create to be out of scope. Furthermore, we are particularly interested in traffic that is related to transactions that could create long lasting harmful effects to the user (e.g., financial loss). Examples of such transactions include:

- Transferring money through an online financial service.
- Modifying text-value fields (e.g., the speed of a turbine, or the water level in a nuclear power plant) of the SCADA (Supervisory Control And Data Acquisition) systems.
- Sending a message through an e-mail client, or an internet messaging (IM) application.
- Posting a status update or comment message through an online social network.

Examples of applications suitable for **GYRUS** include email clients, instant messaging applications, various online social networks, and online financial services. We will further illustrate how **GYRUS** can protect critical actions of these applications in §2.5. Our results indicate that the proposed idea of WYSIWYS is very effective in stopping these applications

from being used to send manipulated traffic by the malware, thus blocking many traditional venues to profit from compromising hosts. In other words, GYRUS can protect sensitive transactions with rich user-generated semantics from malware on the host. For example, GYRUS can prevent botnet malware from sending spam e-mails and instant messaging spam, launching impersonating attacks such as spear phishing, and preventing malware that transfers money from an online banking account.

2.4 Design and Implementation

2.4.1 Architecture

GYRUS employs a virtual machine based isolation mechanism; therefore, its architecture is separated into two parts. GYRUS puts all trusted monitoring modules in either dom-0 or the hypervisor, while dom-U remains untrusted. The architecture of GYRUS is summarized in Figure 3. GYRUS is composed of several key components:

Authorization Database. The *Authorization DB* stores authorization vectors, which contain sufficient information to validate outgoing traffic based on a user's intent. An authorization vector is generated by the *Central Control* and allows us to temporally decouple capturing user intent from the actual enforcement of the WYSIWYS policy at the network interface. At this level, our monitoring is independent of the internal logic of the application. Input events that trigger network traffic (e.g., clicking `SEND` in an e-mail client or pressing the `ENTER` key in the text box of an instant messenger application) will invoke the Central Control to create an authorization vector based on the captured intended content and save it to the authorization database. Later, when the outgoing traffic is generated from the application after processing user input, the traffic will be analyzed in the *Network Monitor*, which will look up the database for evidence of user intent. Our Network Monitor will authorize the traffic only if there exists a matching authorization vector. Otherwise, it will drop the packet. Moreover, this decoupling enables GYRUS to handle asynchronous, or scheduled traffic like e-mail queued to be sent at a later time.

Network Monitor. The *Network Monitor* is a transparent proxy with a built-in monitoring capability. This component inspects all traffic under the monitored protocol. If outgoing traffic is using a protocol corresponding to any of the applications protected by **GYRUS**, the traffic is inspected by querying the Authorization DB to see if the traffic is intended by the user. Unintended traffic is blocked. We also note that the Network Monitor will allow all traffic from other protocols (i.e., not being monitored) to pass through without inspection.

User-Intent Signature. The *User-Intent Signature* captures all the application-specific logic in **GYRUS**. The signatures are expressed in a language that we designed specifically for Gyrus. It covers three categories of information: the condition that triggers network traffic, the required UI structure data for capturing content-intent, and the content of the monitored traffic, which will be matched with UI data.. This user-intent signature language represents our effort to simplify and provide structures to the development of per-application logic under **GYRUS**.

Central Control. The *Central Control* contains the logic that runs the other elements. Its main task is to process intercepted hardware input events. Upon arrival of these events, the Central Control will query the UI monitor to see if the event signifies a user intent to send the currently displayed content out to the network. If so, the Central Control will query the Secure Overlay and the list of active User-Intent Signature to generate an authorization vector for the expected traffic and save it in the Authorization DB. The hardware input event will then be delivered to dom-U, finally reaching its intended destination: a user-driven application. Since the Central Control does not modify any inputs, it does not change the user experience beyond adding an imperceptible delay (see Table 2).

In summary, the workflow of **GYRUS** can be described as follows (Figure 3): The UI Monitor communicates with the Secure Overlay to keep the information displayed in the overlay up-to-date (0). A hardware input event reaches the Central Control (1). Then, the Central Control queries the UI Monitor to see if this input triggers network traffic or not (2). If it does, the Central Control queries the Secure Overlay (3) to create an authorization

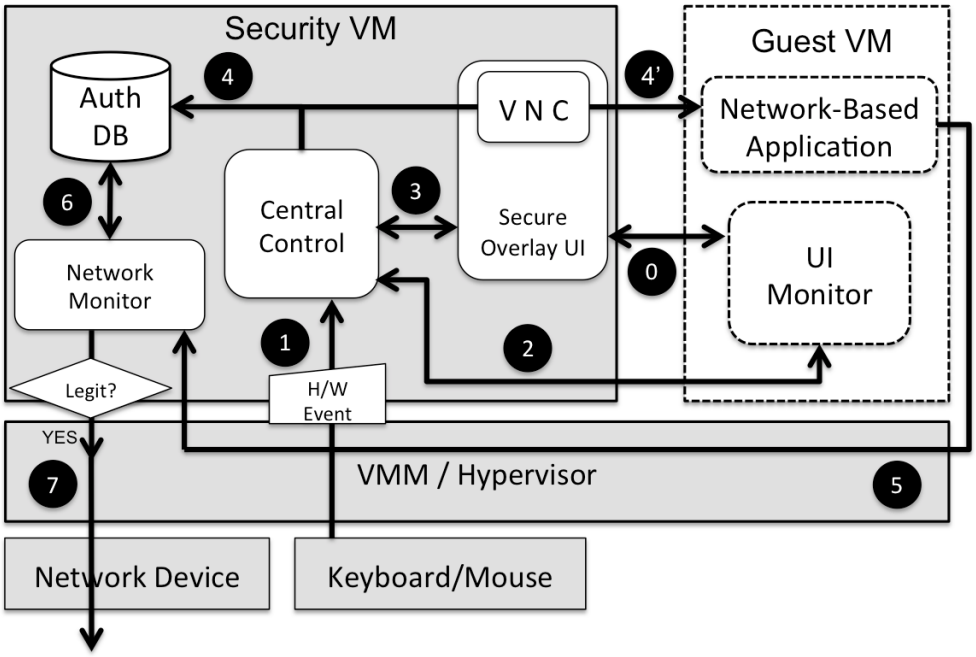

Figure 3: Workflow of Gyrus upon receiving a traffic-triggering event. Grayed and solid-lined areas are trusted components, while dotted lines indicate untrusted components.

vector that describes the user-intended outgoing traffic and save it to the Authorization DB (4). At the same time, the intercepted input event is passed to dom-U (4'). After the application inside dom-U gets the input, it generates the outgoing network traffic (5). The traffic is intercepted and inspected by the Network Monitor. The network monitor then queries the Authorization DB to determine if the intercepted traffic matches user intent (6). The traffic will be allowed to be sent if it matches an authorization vector. Otherwise, traffic is blocked, and **GYRUS** raises the alarm to notify the user of a likely attack attempt (7).

2.4.2 Implementation

We implemented our prototype of the **GYRUS** framework using a Linux/KVM host running Ubuntu 12.04.2 LTS and a dom-U guest virtual machine running Windows 7 SP1. We note that the **GYRUS** architecture is not limited to this particular software stack. We chose KVM and Windows to demonstrate **GYRUS** in a traditional desktop environment. In general, **GYRUS** only requires three platform capabilities: intercepting input & network events,

accessing UI objects, and drawing a secure overlay UI. Therefore, GYRUS could be implemented on a variety of different platforms. For example, GYRUS could use BitVisor [144] as a lightweight secure hypervisor or could use the Dalvik VM [8] on Android as isolation and hardware event-capturing instrument. Similarly, the UI Monitor is not limited to the UI Automation on the Microsoft platforms. Other accessibility frameworks – such as ATK [62] and XAutomation [147] on Linux, and NSAccessiblity [107] on Mac OS X – could replace it. Finally, GYRUS could be implemented using a thin-client model with the trusted client terminal [109] and a network monitor on the remote host.

In-Guest UI Monitor. Since our implementation of the UI monitor is primarily based on the UIAutomation library from Microsoft, we begin with a brief description of this library before presenting details about the UI monitor.

UI Automation. The UIAutomation library represents the UI structure of every window in the system as a tree of UI objects. The root of the tree is the desktop, lower level nodes correspond to individual windows, and further down nodes correspond to components of a window (e.g., buttons, edit boxes, etc.). This tree is similar to the document object model (DOM) tree in a web browser. Each UI object contains data that describe the visual aspects of the corresponding components (e.g., size, visibility, textual content). The UIAutomation library exposes this tree to calling programs through a set of functions that facilitate traversing and querying the tree (e.g., we can search for nodes in the tree with certain properties, or at a particular location on screen), and allows us to access all properties of the nodes. Furthermore, the UIAutomation library also allows calling programs to listen for changes in both the structure of the tree as well as properties of individual nodes.

As mentioned in §2.3, the UI monitor is a component that runs in dom-U, and it serves two purposes: one is to determine if a keyboard/mouse input event [4] signifies the user's intent to send something over the network. And the other is to provide information to the

[4]Input event here is not the real hardware input event. All of the hardware input is handled by Central Control, and the UI Monitor receives a signal from the Central Control when an event arrives.

secure overlay to display up-to-date user generated text in target applications. In other words, implementation of this component needs to provide two primitive operations: identifying the object targeted by an input event and extracting UI properties from text boxes of interest.

Identifying UI Objects. To check if the current input generates network traffic, the UI Monitor first looks for the UI object that receives the current input. For a mouse click event, the UI Monitor calls a Windows API named `ElementFromPoint` to get the object that is currently located under the cursor to determine if the click signifies a user's intent to generate outgoing traffic. For the keystroke events, we use the `GetFocusedElement` API to retrieve the currently focused object (which is also the target of the current input). Upon retrieving the target object for the input event, we can determine if it is a button or a text box of interest by querying the UIAutomation library for the properties of this object. The application-specific logic required for determining the traffic-triggering event is configured in a *User Intent Signature* (e.g. checking if it is a button with its name being `Send` on an e-mail client). Upon receiving an event that generates traffic, the UI Monitor collects UI structure information specified in the User Intent Signature then uses this to inform the Secure Overlay that the traffic-triggering event has occurred. The Secure Overlay also receives the details about what operation and which application triggered the event, and the content from UI data required to generate an authorization vector. A point worth noting here is that we block all updates to the Secure Overlay when we query the UI monitor. This prevents any malicious updates on visible data right before the event, even with the prediction of user's behavior on traffic-triggering event. Additionally, we ensure that the query to the UI monitor completes before the actual input is delivered to the application inside the dom-U, so it will not interfere the application's behavior §2.6.1 presents a more detailed security analysis of **GYRUS**.

Extracting Text and UI Structure Data. To support the Secure Overlay, the UI Monitor needs to extract the user-intended text and associated UI properties. First of all, before extracting the currently displayed text, the UI Monitor registers the text box to the Secure

Overlay to track its properties. Whenever a text box is in focus, the UI Monitor will assign it a unique ID based on the `AutomationID`, an identifier from UIAutomation, of the UI object. This identifier will be used for updating properties of the overlaid text boxes, and indicating which text boxes are needed for generating an authorization vector. At the same time, it extracts the required properties from the text box to support overlaying. To get the screen location of the text box, we query its `BoundingRectangle` property. For text boxes that support properties such as rich text, formatting, text selection, and scrolling, we extract them from the `TextPattern` object. Finally, the UI Monitor captures user-intended text from the `Value` property of a target text box. For text boxes with hidden content (e.g., scrolled-out text), the `Value` and `TextPattern` properties together provide the complete content and useful position information. The Secure Overlay will be notified of all extracted data, along with its identifier, to enable displaying this information back to the user.

To handle updates to the target text box, once we register a text box, we add an event handler to subscribe the `PropertyChangedEvent` of the target object for its `Value` property once we register a text box to the Secure Overlay. In the event handler, we send the updated content to the overlay. This method will update the Secure Overlay whenever the user edits the text. Finally, we register to listen for the change in position of the caret object and forward this information to the overlay so that we can display the caret correctly.

In addition to getting properties for the target text box object, the UI Monitor tracks windowing events when multiple target applications are involved. In particular, we adopted the policy of only displaying the text content of the currently focused window on the overlay; this policy significantly simplifies our implementation and only has a small impact on the usability of our system [5]. Although overlaid text boxes for background applications are not displayed, the Secure Overlay maintains previously captured user-intended text while

[5] Alternatively, we could keep track of the visible region of each target application by implementing a mirror display device driver. We have successfully implemented this functionality, but have not yet integrated it with the rest of our system.

it is visible, and disables its update while it is hidden [6]. Therefore, **GYRUS** can protect the integrity of the content of text boxes in background applications even if it is not shown on the screen. To handle window focus change, we listen for the system-wide `FocusChanged` and `WindowClosed` events from the UIAutomation library. In the handler of these events, we signal the Secure Overlay to hide the content of the window that is closed or has lost focus and to display the content of the newly focused window. We also listen for the `EVENT_SYSTEM_MOVESIZEEND` event and send the Secure Overlay the updated location of the textual content of the target application whenever it is moved or resized. Finally, we choose not to listen for events related to window creation but only handle newly opened applications when the text boxes of interest in these applications first receive focus.

Secure Overlay and Central Control. We implemented both the Secure Overlay and the Central control components as Java programs that run in dom-0. Since the implementation of the Central Control is quite simple, we will not present the details here. However, some implementation details of the Secure Overlay warrant further discussion.

The Secure Overlay has two primary tasks. First, it is responsible for securely displaying the user-generated text, as captured by the UI monitor in dom-U. This part mainly involves some UI/graphics programming and some bookkeeping to group captured text in the same window together for proper handling of windowing events (in particular, when a window gains or loses focus, we need to show or hide all captured text for this window). Our experiments show that the UI monitor provides us with sufficiently rich information to provide a seamless user experience; captured text is rendered without noticeable a difference in terms of location, size, font and color (including background color for highlighting text).

The second task for the secure overlay is to capture and reconstruct the user's intent based on all the textual content that is displayed in the overlay window. By doing this, we can determine what the user-intended outgoing traffic should look like when the user finally

[6] Allowing updates while invisible would prevent reflective verification. If an update is made, the text box will be marked as being "dirty" and will not be used for creating an authorization vector until the user sees the updated content by moving focus into corresponding application

decides to commit what she has typed to the network. Upon receipt of a traffic-triggering event, the UI Monitor will send the tag name of the User Intent Signature, along with identifiers for the text boxes that are required to reconstruct a user's intent to the Central Control. Based on the tag-matching with a User Intent Signature, the Central Control extracts text content for each corresponding text box from the Secure Overlay, builds an authorization vector with them, and saves it to the Authorization DB.

For creating an authorization vector, the Secure Overlay should maintain the user-intended text. In the case where all the user-generated text is displayed on the screen, the Secure Overlay can easily maintain the user-intended text. However, the task is more complicated if the text is displayed in a text box with a scrollbar. In this case, the UI Monitor is still able to capture all the text in the text box; however, reflective verification will not work for the text that has been scrolled out of view. As such, malware in dom-U can modify the invisible parts of the text without the user noticing. To solve this problem, the Secure Overlay keeps track of changes in the content captured by the UI monitor and only considers updates to the target text box that satisfy the following criteria as valid:

- Updates cannot occur at multiple non-consecutive locations (i.e., the difference between the old version and the new version of some captured text can only be the result of inserting or deleting a single character/chunk of text).
- Updates can only occur in the visible part of the text (i.e., the point where the character or chunk of text is inserted or deleted must be visible before the update occurs).
- If a chunk of text is inserted, the end of the chunk must be visible after the update. Similarly, if a character is inserted, the character must be visible after the update.
- If a chunk of text is deleted, the text following the deleted chunk must be visible after the update. Similarly, if one character is deleted, the character that follows must be visible after the update.

If the UI Monitor reports updates that violate the above condition, the Secure Overlay will draw a red border over the corresponding text box to let the user know of the violation.

In this case, the user should check the text displayed by the overlay to determine if GYRUS correctly captured her intent. If it was, she then commit the input to the network. The above design allows us to correctly and securely handle typical operations like typing, deleting text using "backspace", copy-and-paste, deleting/replacing a chunk of highlighted text, even autocomplete and auto-spell-correction. The only caveats we know of are: 1) "Find and replace all", and 2) if the user pastes a chunk of text that is too long to be displayed all at once, some of the pasted text will not be visible in the entire process, and is subject to illegitimate modifications by malware. In these cases, the best practice will be for the user to scroll through the pasted text to ascertain the correctness (and we believe this is a reasonable practice, even if not for security reasons).

Authorization DB. The Authorization DB saves the user intent captured by the Secure Overlay at the time when we capture an input event that signifies the user wants to send something out to the network, and is queried by the Network Monitor when the monitor observes actual outgoing traffic of the corresponding protocol. To allow an efficient lookup, we implement the Authorization DB as a hashtable stored in Ruby, indexed by a data structure called *authorization vector*, which captures both the exact content of the expected outgoing traffic as well as the expected protocol used to send the content. We also associate each key in the hashtable with a numeric value which indicates how many messages matching that key can be sent, so we can handle scenarios where the user intends to send the same message for multiple times.

Network Monitor. The Network Monitor is implemented as a set of transparent proxies, one for each protocol of interest. Each of these proxies has deep packet inspection capability, and we used `iptables` to redirect all of the traffic of each monitored protocols' port to the corresponding proxy for inspection. For SMTP and YMSG, we used stand-alone proxy software proxsmtp [156] and IMSpector [78], respectively. For HTTP, even though there exists a transparent proxy with the capability of ICAP [48] handling such as Squid [158], we

wrote our own implementation because of performance issues [7]. For SSL/TLS encapsulated protocols (e.g., HTTPS, and SMTP TLS), we use the Man-In-The-Middle (MITM) approach to decrypt the traffic to be analyzed, and re-encrypt it afterward. In particular, we created a self-signed CA certificate and CA-signed wild-card certificate, and inject the CA certificate to dom-U as a trusted CA. With these certificates, **GYRUS** can impose itself as the server at the setup phase for SSL connections, and be able to decrypt any subsequent traffic from dom-U to the actual server. Finally, we note that this MITM approach is not an invention of our own, but is widely used approach for deep packet inspection (DPI) with various intrusion detection/prevension systems (IDSs/IPSs) [140].

User-Intent Signature. As we have mentioned in §2.1, an approach that tries to model and enforce correct behavior of applications will inevitably have some per-application logic. To make this development process as painless as possible, we created our own language for specifying the per-application logic as well as the programs to interpret the specifications. We call specifications under our language *User-Intent Signatures*, and we express these signatures in the JSON (JavaScript Object Notation) format. Each user intent signature contains eight JSON object fields, and the names of the fields are: `TAG`, `WINDOW`, `DOMAIN`, `EVENT`, `COND`, `CAPTURE`, `TYPE`, and `BIND`. In the following, we will give a brief description of each with its intended purpose. Please refer to Figure 5, and Figure 6 for examples of user intent signature as well as more specifics of the signature language. Before starting, we first note that the `TAG` field in this signature is for assigning a unique signature name.

Identifying Traffic Event and Focused Application. Our monitor component, the UI Monitor, uses this signature to identify traffic-triggering input events. To specifying a traffic-generating event in a User Intent Signature, the signature writer can set the `EVENT` field. This field will contain the value of required hardware input event. For example, it could be `LCLICK` to indicate a left mouse click on the send button of an e-mail client or

[7] Because Squid does not support multi-threading for traffic relaying, it can cause severe delays when a web browser loads a web page.

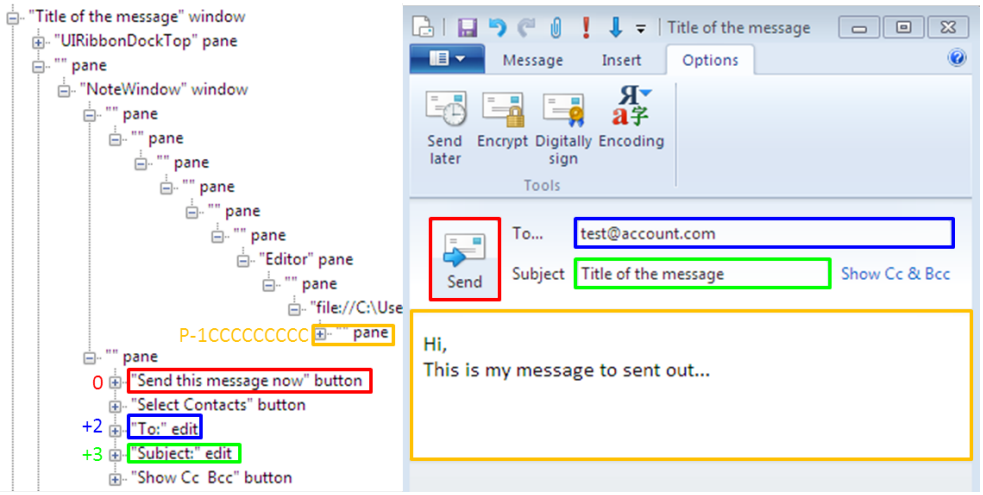

Figure 4: UI structure of Windows Live Mail. Tree structure on the left is from `Inspect.exe`. `0` indicates event-receiving object (send button), `+2` and `+3` indicate the 2nd and the 3rd sibling from the origin (a negative number indicates the previous sibling). `P` is a symbol for a parent, and `C` refers to a child.

`ENTER` for reacting on pressing the return key on the message dialog of an instant messenger application. This field only partially defines the traffic-triggering event. It should be linked with application-specific, operation-specific information to correctly identify if the input will be delivered to the specified application. For correctly figuring out the details from an input receiving application, we use the tree-structure information of UI from UI Automation, in addition to a simple indicator such as the name of the window for stand-alone apps or domain name of currently visiting page URL in web-apps. Figure 5 shows how the signature is constructed to detect a Windows Live Mail application. In the compose view of the application, its window name is always `ATH_Note`, so the `WINDOW` field indicates this information. The sub-components in the UI tree-structure are – starting from the event receiving object – a button named "Send this message now", text edit boxes for the `To` and `Subject` fields, and a content pane for the e-mail message text. Figure 4 illustrates how UI Automation manages this tree-structure. Note that the `COND` section lists all of the conditions of the tree-structure. For internal fields, the number indicates the relative distance from an event receiving object as a sibling on the tree. So `0` means the object specified triggered the event, and `+2` or `+3` indicates the next siblings at the specified distance (a

```
{
  "TAG" : "LIVEMAILCOMPOSE",
  "EVENT" : "LCLICK",
  "WINDOW" : "ATH_Note",
    "COND" : {
      "0" : {
        "CONT" : "BUTTON",
        "NAME" : "Send this message now"
      },
      "+2" : {
        "CONT" : "EDIT",
        "NAME" : "To:"
      },
      "+3" : {
        "CONT" : "EDIT",
        "NAME" : "Subject:"
      },
      "P-1CCCCCCCCC" : {
        "CONT" : "PANE"
      }
    },
    "CAPTURE" : {
      "A" : "+2.value",
      "B" : "+3.value",
      "C" : "P-1CCCCCCCCC.value"
    },
    "TYPE" : "SMTP",
    "BIND" : {
      "METHOD" : "SEND",
      "PARAMS" : {
        "to" : "A",
        "subject" : "B",
        "body" : "C"
      }
    }
}
```

Figure 5: User Intent Signature for sending e-mail on Windows Live Mail.

```
{
  "TAG" : "FBCOMMENT",
  "EVENT" : "ENTER",
  "DOMAIN" : "www.facebook.com",
  "COND" : {
    "0" : {
      "NAME" : "Write a comment...",
      "CONT" : "EDIT"
    },
    "P-1" : {
      "CONT" : "IMG"
    }
  },
  "CAPTURE" : {
    "A" : "0.value"
  },
  "TYPE" : "WEB",
  "BIND" : {
    "URL" : "www.facebook.com/ajax/ufi/add_comment.php",
    "METHOD" : "POST",
    "PARAMS" : {
      "comment_text" : "A"
    }
  }
}
```

Figure 6: User Intent Signature for posting comments on Facebook Web-app.

negative number indicates the previous sibling at the distance). `P` and `C` refers to a parent and a child, respectively.

For the UI Monitor, when an input comes, we iterate over all signatures that have an `EVENT` field with the current input, and check the UI tree-structure conditions to determine whether current input triggers traffic or not. If it does, as our workflow goes, the required data will be sent to Central Control to generate an authorization vector.

The Network Monitor also uses this signature for determining whether the current packet is monitored or not. The `TYPE` field specifies the monitored protocol. Its value can be a protocol name (e.g., SMTP for e-mail client and WEB for web-apps). Since network monitor only traps some transactions for each protocol, to bind a signature to a certain transaction, we use the `METHOD` field under `BIND` to specify the desired transaction for non-web protocols [8], and both the `METHOD` and the `URL` [9] fields are used for web-apps (`METHOD` is for distinguishing

[8] We assigned natural names for each operation. The line `"METHOD" : "SEND"` in Figure 5 means that the signature should only monitor the sending operation in the SMTP protocol.

[9] Similar to the remote procedure call, a URL in a web-app is analogous to invoking a function on the host, so a URL can indicate a particular transaction.

GET and POST messaging in web-apps).

Specifying User-Intended Text. The User Intent Signature is also responsible for indicating which text boxes correspond to the user's intent, for generating authorization vectors. With the UI Monitor, it uses the CAPTURE field to indicate text boxes that contain user-intended text. In this field, the left-side key value is assigned alphabetically to simplify text matching in for network packets, and the right-side indicates the location of the target text box on the UI tree-structure and any required properties for it. According to this information, the UI Monitor transmits a unique identifier of target text boxes to the Central Control, and then the Central Control extracts a user verified text from the Secure Overlay, and then the Central Control finally creates an authorization vector. The vector will be in a form that can be reconstructed within the Network Monitor.

For the Network Monitor, it refers to the PARAMS field to extract content from the packet. The left-side key value for this is a natural name for the stand-alone protocol or the URL parameter for web-apps. The right-side value has an alphabet value that is previously assigned in the CAPTURE field, which is used to link captured text boxes to each parameter within the current packet. Since an authorization vector is created with the knowledge of the PARAMS field, the Network Monitor can reconstruct the correct vector using only this packet and signature data. After reconstructing the vector, we query the authorization DB to check for proof of previously established user intent.

2.5 Application Case Studies

In this section, we will present our experience in using **GYRUS** to protect existing applications. Our experiments cover traditional, stand-alone applications as well as web applications. For stand-alone applications, we studied how to apply **GYRUS** to Windows Live Mail and Digsby (an instant messaging client). For web applications, we picked the following from the top 25 sites according to Alexa [3]: GMail, Facebook and Paypal, and for our studies, we assume these web applications are accessed using Microsoft Internet Explorer 10. We

argue that these applications represent some of the most important ones in daily life. We base this argument on the Pew Internet survey called "What Internet Users Do On A Typical Day" [130], which lists sending/reading emails, using online social networking, doing online banking, and sending instant messages among the 20 things most people do on a daily basis. We also observe that the remaining of the listed popular activities mostly involve users getting information from the Internet, and does not require the transmission of any user generated content, and thus are not the target for **GYRUS** protection.

The focus of the following discussion is on how we can specify the per-application logic necessary for **GYRUS** protection for each of the target applications using a User Intent Signature. We believe our experience shows that the User Intent Signature language makes this task very manageable.

Constructing User Intent Signature. Construction of a User Intent Signature is two folded as **GYRUS** decouples capturing of the user intent and monitoring of the network traffic. The UI part of the signature can be constructed intuitively. First, we arrange the UI as it would be used for composing user-generated content. Then we identify an input event that triggers traffic and the associated text boxes that contain user-intended text through a visual inspection of the UI. Next, with the help of a tool called `inspect.exe` from the UI Automation library, we can identify the tree-structure and other details of the UI. Finally, this information is used to construct the definition distinguishing the application that receives input events.

The second part, the network side, requires an understanding of the underlying protocol that the application uses for network communication. In particular, we need to identify which traffic we should intercept for monitoring, and discover how the user-intended text is formatted within the packet. In this section, we provide examples of applications that can be protected by **GYRUS**, and we demonstrate how the User Intent Signature simplifies the process for supporting a new application.

2.5.1 Windows Live Mail

Application Specification. Windows Live Mail is a stand-alone email client, and the focus of our experiment is to use GYRUS to make sure that any outgoing e-mail messages (i.e., through SMTP) are intended by the user. The user interacts with a compose window to write a message. The window has a `Send` button that user will click when the user decides to send the message. And there are several text boxes reserved for a list of recipients (e.g., `To`, `Cc`, etc.), and the message Subject. Finally, the window has a rich text pane at the bottom, to compose the content of the message.

Event and Intended Text. The traffic will be generated after the user clicks the `Send` button. On the event, GYRUS will extract user-intended texts from the `To`, `Subject`, and the message body text pane.

Network Traffic Specification. Outgoing traffic will be sent through the SMTP protocol, and we are specifically interested in the portion of the SMTP exchange responsible for sending a message. All user-generated text will directly be shown as the same text in the traffic, and GYRUS will extract each field to query to the Authorization DB.

Constructing Signature. We show the signature for this application in Figure 5. The input event that triggers traffic creation is pressing the `Send` button. So we set the `EVENT` field to `LCLICK`. To distinguish the application window, we set `WINDOW` field to the classname of the e-mail composing window, which is `ATH_Note` in this case. To improve the event condition that detects the application, we list all UI objects that are required to capture user intent in `COND` section. Starting from the event receiving object, the `Send` button, the text box for recipients is the second sibling, and the text box for the subject is the third sibling. So we mark them as `+2`, `+3`, respectively. Locating the rich text pane used for the message also requires tree-traversal. In our scheme, it is located at `P-1CCCCCCCCC`. Since we need to capture the contents of all text boxes and the pane, in the `CAPTURE` field, we assign temporary variables to each UI object as `A`, `B`, and `C`.

For the network monitor, we set the protocol by assigning `SMTP` in the `TYPE` field, and `SEND` in the `METHOD` field, and bind each of the variables assigned during the `CAPTURE` stage to protocol specific variables.

2.5.2 Digsby: Yahoo! Messenger & Twitter

Application Specification. Digsby is a stand-alone client for accessing multiple instant messengers and online social network services within one application. In our experiments, we focus on using **GYRUS** to protect the outgoing communication to Yahoo Messenger and Twitter. Communications to other messengers/online social network services can easily be covered as long as we have the corresponding proxy for handling the network traffic. We would simply require one user intent signature for each supported protocol. For both Yahoo Messenger and Twitter, Digsby provides a simple GUI. The user interacts with a messaging dialog window, which has a text box for the message at the bottom. After typing a text message, the user can send the message by pressing the `ENTER` key while still focused in the message text box.

Event and Intended Text. The traffic will be generated after pressing the `ENTER` key. At this time, **GYRUS** will extract user-intended text from the message text box at the bottom of the dialog.

Network Traffic Specification. For Yahoo Messenger, outgoing traffic will be sent through the Yahoo! Messenger (YMSG) protocol. Similar to the e-mail case, we are only interested in the portion of the protocol that contains the message. The user-intended text will be encapsulated with HTML tags for formatting, so **GYRUS** will extract the text and then query the authorization DB. For Twitter, Digsby will communicate with its server through an HTTP REST API. The network monitor needs to watch for POST requests to `https://api.twitter.com/1/statuses/update.json`. In this case, the user-intended text will be encoded with URL encoding, so the extracted text will be queried to the authorization DB after proper decoding.

Constructing Signature. Pressing the ENTER key after typing a message triggers outgoing network traffic. Looking up the class name of the dialog window, Digsby uses wxWindowClass for Yahoo Messenger and wxWindowClassNR for Twitter. To improve the event conditions of the UI structure, in addition to checking whether current input is delivered to the text box for a message, we also check if it has a pane object as its siblings. Since we need to capture user-intended text from the message text box, we assign the variable A to it for the CAPTURE field. On the network side, we set the protocol type as YMSG and WEB respectively. We set the METHOD field as SEND for YMSG, and POST for Twitter. Variable A for the intended text will be bound to a variable called message in YMSG, and status for Twitter.

2.5.3 Web-App: GMail

Application Specification. The workflow of GMail is very similar to that of Windows Live Mail. It has a Send button on top of the compose screen[10], along with To, Subject, and the message pane.

Network Traffic Specification. On clicking the Send button, an e-mail message will be sent through a POST method URL https://mail.google.com/mail. The GMail application accesses the URL for multiple purposes, however, the URL only serves for sending an e-mail message when the URL set with a URL parameter act=sm. The user-intended text is transmitted in the to, subject, and body parameters of the POST request.

Constructing Signature. Because the left-click on the mouse is the traffic-triggering event, we set the EVENT field as LCLICK. For the application UI condition, we use the domain name mail.google.com as a window identifier, along with the relative positions of text boxes to the Send button. For the network traffic, the trap condition is the URL https://mail.google.com/mail with parameter condition act=sm. At the Network Monitor, UI variables in the CAPTURE field will be matched with POST parameters named

[10] We ran **GYRUS** with the old version of GMail composing UI, which was available until July 2013.

to, subject and body.

2.5.4 Web-App: Facebook

Application Specification. We focus on three transactions in the Facebook application: posting a status update, posting a comment, and sending a message. For the status update, the user types a message in the text box and clicks the Post button. This is similar to the e-mail applications. For adding a comment and sending a message, the user presses the ENTER key after composing her message, which is analogous to the Digsby example.

Network Traffic Specification. For the status updates, traffic goes to https://www.facebook.com/ajax/updatestatus.php, and the user-intended text is transmitted in the POST variable xhpc_message_text. Adding a comment goes to https://www.facebook.com/ajax/ufi/add_comment.php, and the user-intended text is transmitted in the POST variable comment_text. Finally, sending a message goes to https://www.facebook.com/ajax/mercury/send_messages.php, and the user-intended text is transmitted in the POST variable message_batch[0][body].

Constructing Signature. The traffic-triggering event is LCLICK for status updates, and ENTER for the others. Identifying the application and expected transaction for each event is challenging because all three transactions are done in the same window so we cannot distinguish each transaction using only the domain name. Therefore we can distinguish each transaction using additional UI structure checks. We link camera, location, and emoticon menu icons as siblings for distinguishing status update, link profile image and the shadow text "Write a comment" for adding a comment, and link an icon name with "Add more friends to chat" and conversation history objects as siblings for sending a chat message.

2.5.5 Web-App: Paypal

Application Specification. Using GYRUS with Paypal enables a validation of the integrity of the amount of money sent to someone. On the "transferring money" page, after he types

the username or e-mail address of the recipient and the amount of money to transfer in the text boxes, the user clicks the `Continue` button to send the money. The workflow is analogous to our e-mail examples, with the primary difference being that the message is the amount of money to be transferred in this case.

Network Traffic Specification. After clicking the `Continue` button, the traffic will be sent to `https://www.paypal.com/us/cgi-bin/webscr` if its POST parameter has the following value: `cmd=_flow`. The user-intended text will be placed in the POST parameters, `email` and `amount`.

Constructing Signature. The event is set to `LCLICK`, and the application condition will check domain `www.paypal.com` and whether the UI tree-structure has all participating text boxes as siblings. The traffic trap condition for the Network Monitor should be set as `https://www.paypal.com/us/cgi-bin/webscr`, and its POST parameter named `cmd=_flow`. Finally, variables for captured text for the amount of money and the recipient will be linked to the POST parameters `amount` and `email`, respectively.

2.5.6 Discussions

In cases where multiple applications use the same protocol, we need one User-Intent Signature for each application using that protocol. While this may seem a lot of work, defining the correct behavior of applications of interest is still much more scalable than endlessly (re)modeling (new) attack/malware behavior.

As we've shown in the examples above, the language we have devised not only allows us to support new applications easily but it also cleanly separates the per-application logic from the core **GYRUS** framework. With this language, the process of specifying the User Intent Signature for an application only requires knowledge about the UI (and the structure of the UI object tree exposed by the UI Automation library for that application, which can be obtained using standard tools like Inspect [119] from Microsoft) and some knowledge about the network protocol used by the application, but no further details about the internals

Table 1: List of activities where Gyrus can help to protect the corresponding network transactions, from the survey 'What Internet Users Do Online [131]', by Pew Research Center.

Activity	% of Users
Send or read e-mail	88
Buy a product	71
Use a social networking site	67
Buy or make a reservation for travel	65
Do any banking online	61
Send instant messages	46
Pay to access or download digital content	43
Post a comment to online news groups	32
Use Twitter	16
Buy or sell stocks, bonds, or mutual funds	11

of the application (as compared to if we used VM introspection techniques to extract user intent).

Although it is easy to construct a signature for supporting a new application, managing a large collection of signatures could cause overhead. However, we argue that its overhead is far less than that of traditional IDS and anti-virus software. While traditional approaches require following up all newly discovered attacks, GYRUS defines user-intended, correct system behaviors and is therefore attack-agnostic.

The term attack-agnostic here does not mean that GYRUS is immune to all kinds of attacks. GYRUS only makes assumptions about the attacker's goal, but not how they achieve this goal. That is, once a user intent signature is defined, no matter how the attack evolves, the protection mechanism of GYRUS still works. In this work, we focus on protecting the integrity of text content that is typed by the user, while other kinds of attacks such as confidentiality of data are out of scope.

Regarding application support, GYRUS can generally support any application that sends user-generated text content from the monitored host, if its network traffic has a direct or simple mapping with on-screen text content. Table 1 shows the result of a survey that indicates what typical users are doing on the Internet, done by Pew Internet. According to

the survey results, 88% of users send e-mail, 67% of them send their text content to online social network (OSN) sites, and 61% of users use online banking. Moreover, all activities listed in Table 1 can be supported by GYRUS. Clearly, GYRUS can protect a significant portion of day-to-day user activities on the Internet and can have a large impact on security.

While the focus of GYRUS is text-based applications, it can be easily extended to handle image/video attachments. In particular, GYRUS can adopt Access Control Gadgets [133] to capture the user's intent to attach a particular file, compute a checksum of that file and have our network proxy match any attached file against the checksum. The only way this mechanism would fail is when an attacker/malware knows a priori which file the user will attach and changes it in advance, which we consider being unrealistic.

One limitation of GYRUS is that it cannot protect an application where the user-intended text is represented in a proprietary format or in some complicated encoding on the traffic. At least, not without significantly more effort to reverse engineer the format. This can be a problem when extending GYRUS to more general transactions such as writing data on the filesystem. There have been recent and promising advances in verifiable computation and tools such as probabilistically checkable proofs (PCP) and fully-homomorphic encryption (FHE) are becoming practical. When these technologies come to practice, GYRUS can verify if the result of the traffic is actually from the user-intended input, by running application logic along with these computation proof mechanisms.

Additionally, for applications with complex encodings mentioned above, we believe that it would be possible to have GYRUS perform the slightly more complicated transformation on the captured user intent and match the result with the outgoing traffic. Though we should be careful not to expand the TCB too significantly, adding the support of the specific transformations of some of the most popular applications should be quite doable.

In our threat model, GYRUS only protects the integrity of the text based on a user's intent, and it does not protect confidentiality. An attacker could steal a user's credentials (e.g., Cookie and ID/Password), and then perform protected transactions on a different

host without GYRUS protections. Thus, GYRUS works better when the host is equipped with Hardware Security Module (HSM) such as Trusted Platform Module (TPM) and a Smartcard, and the server-side of the application supports mutual authentication. However, while we consider the defense against stealing credential to be out of scope, we would point out that this problem can be solved by using GYRUS. The way is, in the dom-0, GYRUS can intercept and modify the password that the user has just entered (and so malware in dom-U can only get the incorrect password), and correct the subsequent outgoing traffic for the actual login to use the unmodified, correct password.

Finally, like any other system that tries to model benign behavior, GYRUS is vulnerable to false positives caused by errors in the user intent signatures (false negatives are also possible, but should be a lesser concern, as we will argue in the next section). However, false positives only happen when we fail to specify in our signatures some of the user actions that signifies the intent to generate outgoing traffic, or if our signatures specify a wrong way for capturing user intent. We believe both scenarios should be rare because an application should not have too much variance in its UI nor should it provide too many ways for performing the same operation for the sake of application's usability. Similarly, the correctness of the way we capture user intent for an application should be easy to establish with simple testing, and this should suffice to guarantee that we will continue to capture user intent correctly unless the application changes its UI (which, again, for usability reasons, is less likely to happen).

2.6 Evaluation

In this section, we present the results of our evaluation of the security, usability, and performance of GYRUS when using it to protect the applications studied in §2.5.

2.6.1 Security

New security frameworks should be secure against both current and future attacks. Here, we consider both scenarios for GYRUS by running existing attack samples and by analyzing the framework's security properties.

Resilience Against Existing Attacks. GYRUS is attack-agnostic by design, however, to demonstrate that we implemented the system correctly, we tested GYRUS' ability to stop attacks against the specific applications discussed in §2.5. For Windows Live Mail, we executed `Win32:MassMail-A`, a mail spammer malware, while the mail client is under GYRUS' protection. The dom-0 network monitor successfully catches and blocks all outgoing SMTP traffic generated by the malware. For Yahoo! Messenger protocol, we ran `ApplicUnwnt.Win32.SpamTool.Agent. BAAE`, a messenger spamming malware. GYRUS blocks all of the messages generated by this malware. For Facebook, we executed a comment spamming malware, `TROJ_GEN.RFFH1G1`, and it has no success in sending out attack traffic. We have also tested the effectiveness of GYRUS against Javascript-based attacks (like XSS, CSRF) targeting web applications. In particular, we injected forged Javascript code that automatically submits malicious content into the GMail, Facebook and Paypal pages. In all cases, GYRUS successfully blocked all malicious traffic from these attacks. Finally, for each tested application, we tried to perform the normal operations protected by GYRUS with the corresponding attacks running in the background. In each case, Gyrus allows the legitimate, user generated traffic to go through while stopping all attacks.

Resilience Against Future Attacks. Next, we will evaluate how well GYRUS can handle future attacks designed against it. All security guarantees will be void if assumptions in our threat model are violated. However, we believe those are standard assumptions widely accepted by the security community. Thus we will not discuss violations of the assumptions. However, we do note that even though existing hypervisors are becoming more complicated, it is possible – and, in fact, encouraged – to build custom hypervisors or security operating systems for use with GYRUS to achieve higher assurance [96, 35, 162].

The next avenue for attack is the UI monitor that runs in the untrusted dom-U. However, we believe GYRUS is quite robust against errors in the UI monitor. First of all, because of the protections provided by the Secure Overlay, attackers are limited to misplace user-generated, albeit unintended content in traffic allowed by GYRUS (e.g., switching the subject

and content of an email message, take user's comment to one story on Facebook as his/her outgoing comment on another story, etc.). Secondly, our policy of only displaying (on the overlay) the content of the window which currently has focus, the mistakenly sent out content must be from the "correct" application. Additionally, we believe that we can further harden **GYRUS** against such attacks by specifying a restriction on the position of the content to be sent out in relation to the event that triggers the outgoing traffic (e.g., the text displayed on the overlay cannot be too far away from the coordinate of the mouse click). A compromised UI monitor can also mislead **GYRUS** to believe a mouse click signifies the user's intent to send out something (i.e. stealing a click). However, once again thanks to the Secure Overlay, the unintended outgoing traffic will have its content entirely entered by the user (i.e., this could cause a premature output of the content). Therefore, the attacker will have very little control over what is sent. Finally, we believe that our policy concerning what kind of update to text boxes the UI monitor can report provides very good protection to data that are currently off-screen.

Similarly, poorly written user intent signatures can be problematic. However, thanks to the use of the Secure Overlay, we believe problems with a user intent signature are limited to mistaking hardware events as user intent to send something and will have the same adverse effect as a misbehaving UI monitor stealing a click. In conclusion, we believe the Secure Overlay (and the WYSIWYS policy) leaves an attacker with very limited options for attacking **GYRUS**. Anything sent out by a protected application using a targeted protocol must be typed, and seen by the user. All the attacker can do is to use content intended for one purpose (under the same application) for another, and the cases where this can cause a user any real harm should be very rare.

2.6.2 Usability

From our experience of protecting the applications studied in §2.5, **GYRUS** has no noticeable effect on their usability. In **GYRUS**, user-interaction is mediated by the internal components

of GYRUS: Input handler and Secure Overlay. For interposing user input before delivering it to the application, GYRUS does not incur noticeable delay (see Table 2.6.3 for the evaluation results).

Since GYRUS only overlays text boxes in our target applications, it will not change the user's workflow or the look-and-feel of the other parts of the application. Furthermore, GYRUS displays (on the secure overlay) text with the same font face, size, and color as the underlying application. Finally, we have confirmed that the edit box drawn by GYRUS supports not only simple text editing like typing, selection, and copy & paste, but also application-specific text editing features like auto-completion and spelling correction. So we are confident that GYRUS will not affect the user's experience with the application being monitored. Furthermore, since GYRUS only checks (and potentially blocks) traffic that perform specific actions under specific protocols of interest [11], our experience shows that GYRUS does not interfere with background networking programs such as BitTorrent and RSS feeds. GYRUS can also handle scheduled jobs that have a time gap between a user's interaction and the resulting generation of network traffic, thanks to our use of the Authorization DB for the capturing of user intent from the actual inspection of traffic. For example, in the case of an e-mail application, if the system has no connectivity to the Internet, the mail will be queued on the scheduler, and later this scheduler will generate network traffic when connectivity is re-established. Our experiments show that GYRUS can handle this situation correctly and allow the delayed email as a user would expect.

2.6.3 Performance

In this section, we present our results of measuring the two kinds of delay that GYRUS can cause: delay in processing user input through the keyboard/mouse and delay in sending out network traffic. We performed all the experiments presented in this section on a commodity laptop: a Lenovo Thinkpad-T520, equipped with a dual-core Intel Core i5 2520m and 8GB

[11] For example, GYRUS only checks HTTP traffic for sending emails under GMail, but not that for reading emails.

Table 2: Latency introduced by Gyrus while processing the input. The data for user-interaction was collected during the use case evaluation.

Actions	Average	STDV	Median	Max
Typing	39ms	21ms	34ms	128ms
ENTER	19ms	6ms	17ms	43ms
LCLICK	43ms	15ms	41ms	79ms
Focus Change	21ms	19ms	17ms	158ms
Move & Resize	21ms	16ms	16ms	85ms

of RAM. The dom-U runs 3 logical cores with 7GB of RAM, while dom-0 has 1 logical core and 1GB of RAM.

Interaction Overhead. In the worst case, the user will experience the following delay for every keyboard/mouse input:

- The Central Control will need to query the UI monitor in dom-U to see if this event signifies the user's intent to send out something.
- The Secure Overlay will have to wait for the UI monitor to provide any information about how this input changes the display.

Both of these will add to the time from the user press a key/click to mouse to when he/she can see the effect of his/her input on the secure overlay.

To determine if this turn around time for processing user input under **GYRUS** is still in acceptable range, we performed the following study:

- First, we typed a document without generating any input that signifies an intent to send out network traffic, and measured the time from the Central Control first observe each input to the time the Secure Overlay is updated to reflect the input.
- Second, we measured the same turn around time for mouse events that result in focus change, resize and movement of the window of a target application.
- Finally, we also measured the time needed for the UI monitor to confirm that an input event signifies user intent to send out traffic.

Table 2 shows the results of our experiments. To provide some context for interpreting the

Table 3: Network latency for HTTP connections.

Cases	KVM	Gyrus	Overhead
Single (A)	101.7ms	102.3ms	+0.6ms (.5%)
Single (B)	31.20ms	32.30ms	+1.1ms (3.5%)
Web Page	897.5ms	951.3ms	+53.8ms (6%)
Download	51.1MB/s	49.3MB/s	–1.8MB/s (3.5%)

results, we note that prior research suggests that acceptable range of such turn around time for interaction with the human is 50-150ms [145]. Thus, our experiments show that on average case, users can smoothly interact with a system protected by GYRUS.

Network Latency. We have also measured the network latency caused by GYRUS (as compared to the system that runs KVM without GYRUS) for three different cases:

- The time to establish an HTTP connection (and we used two test sites),
- The time to load a web page with dynamic content, measured by The Chromium's Page Benchmark extension [149],
- The effective bandwidth of a system, obtained by measuring the time to download a 550MB disk image from the Debian repository through HTTP.

To measure the overhead introduced by our Man-In-The-Middle (MITM) proxy for HTTPS connections, we did two tests:

- Download 15KB of web-page data from a public website, and
- Download a 32MB file from a remote HTTPS server.

We repeated all experiments for ten times, and Table 3 and Table 4 shows the average results of the experiments.

Comparing the results from a KVM Guest versus GYRUS running on it, GYRUS only introduces around 1 ms of single response delay, less than 6% (53.8 ms) of delay for web page loading, and less than 4% overhead on the network bandwidth, for HTTP connection. For HTTPS, there exists CPU time overhead from an additional connection per each session for MITM on establishing, encrypting, and decrypting the contents. From our experiment,

Table 4: Network latency for HTTPS connections (with Man-In-The-Middle proxy).

Cases	KVM	Gyrus	Overhead
Single Request	90.72ms	94.50ms	+3.78ms (4%)
Download	37.40MB/s	35.23MB/s	–2.17MB/s (5.8%)

it incurs 4 ms of delay on getting access to a single webpage data, and adds less than 6% of bandwidth overhead on downloading of file content. Evaluation results for the network latency show that GYRUS has very little overhead, at worst 6% on both bandwidth and loading a webpage.

2.7 Summary

In this chapter, we introduced the GYRUS framework and showed how it can be used to distinguish between human and malware generated network traffic for a variety of applications. By combining the secure monitoring of hardware events with an analysis leveraging the accessibility interface within dom-U, we linked human input to observed network traffic and used this information to make security decisions. Using GYRUS, we demonstrated how to stop malicious activities that manipulate the host machine to send malicious traffic, such as spam, social network impersonation attacks, and online financial services fraud. Our evaluation demonstrated that GYRUS successfully stops modern malware, and our analysis shows that it would be very challenging for future attacks to defeat it. Finally, our performance analysis shows that GYRUS is a viable option for deployment on desktop computers with regular user interaction. GYRUS fills an important gap, enabling security policies that consider user intent in determining the legitimacy of network traffic.

CHAPTER III

MIMESIS AEGIS: A MIMICRY PRIVACY SHIELD

3.1 Motivation

A continuously increasing number of users now utilize mobile devices [151] to interact with public cloud services (PCS) (e.g. Gmail, Outlook, and WhatsApp) as an essential part of their daily lives. While the user's connectivity to the Internet is improved with mobile platforms, the problem of preserving data privacy while interacting with PCS remains unsolved. In fact, news about the US government's alleged surveillance programs reminds everybody about a very unsatisfactory status quo: while PCS are essentially part of everyday life, the default method of utilizing them exposes users to privacy breaches, because it implicitly requires the users to trust the PCS providers with the confidentiality of their data. But such trust is unjustified, if not misplaced. Incidents that demonstrate a breach of this trust is easy to come by:

- PCS providers are bound by law to share their users' data with surveillance agencies [28],
- it is the business model of the PCS providers to mine their users' data and share it with third parties [15, 2, 17, 51],
- operator errors [43] can result in unintended data access, and
- data servers can be compromised by attackers [74].

To alter this undesirable status quo, solutions should be built based on an updated trust model of everyday communication that better reflects the reality of the threats mentioned above. In particular, new solutions must first assume PCS providers to be untrusted. This assumption implies that the PCS providers control all other entities, including the apps that users installed to engage with the PCS, must also be assumed untrusted.

Although there are a plethora of apps available today that come in various combinations of look and feel and features, we observed that many of these apps provide text communication services (e.g. email or private/group messaging categories). Users can still enjoy the same quality of service[1] without needing to reveal their plaintext data to PCS providers. PCS providers are essentially message routers that can function normally without needing to know the content of the messages being delivered, analogous to postmen delivering letters without needing to learn the actual content of the letters.

Therefore, applying end-to-end encryption (E2EE) without assuming trust in the PCS providers seems to solve the problem. However, in practice, the direct application of E2EE solutions onto the mobile device environment is more challenging than initially thought [160, 142]. A good solution must present clear advantages to the entire mobile security ecosystem. In particular it must account for these factors: 1) the users' ease-of-use; hence acceptability and adoptability; 2) the developers' efforts to maintain support; and 3) the feasibility and deployability of the solution on a mobile system.

From this analysis, we formulate three design goals that must be addressed coherently:

1. For a solution to be secure, it must be properly isolated from untrusted entities. It is evident that E2EE cannot protect data confidentiality if plaintext data or an encryption key can be compromised by architectures that risk exposing these values. Traditional solutions like PGP [148] and newer solutions like Gibberbot [61], TextSecure [126], and SafeSlinger [52] provide good isolation, but force users to use custom apps, which can cause usability problems (refer to (item 2)). Solutions that repackage/rewrite existing apps to introduce additional security checks [166, 20] do not have this property (further discussed in §3.2). Solutions in the form of browser plugins/extensions also do not have this property (further discussed in §3.2), and they generally do not fit into the mobile security landscape because many mobile browsers do not support extensions [32], and mobile device users do not favor using mobile browsers [22]

[1] the apps' functionalities and user experience are preserved

to access PCS. Therefore, we rule out conventional browser-plugin/extension-based solutions.

2. For a solution to be adoptable, it must preserve the user experience. We argue that users will not accept solutions that require them to switch to different apps to perform their daily tasks. Therefore, simply porting solutions like PGP to a mobile platform would not work, because it forces users to use a separate and custom app, and it is impossible to recreate the richness and unique user experience of all existing text routing apps offered by various PCS providers today. In the context of mobile devices, PCS are competing for market share not only by providing more reliable infrastructure to facilitate user communication but also by offering a better user experience [83, 141]. Ultimately, users will choose apps that feel the most comfortable. To reduce interference with a user's interaction with the app of their choice, security solutions must be retrofittable to existing apps. Solutions that repackage/rewrite existing apps have this criterion.

3. For a solution to be sustainable, it must be easy to maintain and scalable: the solution must be sufficiently general-purpose, require minimal effort to support new apps, and withstand app updates. In the past, email was one of the very few means of communication. Protecting it is relatively straightforward because email protocols (e.g. POP and IMAP) are well defined. Custom privacy-preserving apps can therefore be built to serve this need. However, with the introduction of PCS that are becoming indispensable in a user's everyday life, a good solution should also be able to integrate security features into apps without requiring reverse engineering of the apps' logic and/or network protocols, which are largely undocumented and possibly proprietary (e.g. Skype, WhatsApp, etc.).

This chapter introduces *Mimesis Aegis (***M-AEGIS***)*, a privacy-preserving system that mimics the look-and-feel of existing apps to preserve their user experience and workflow

on mobile devices, without changing the underlying OS or modifying/repackaging existing apps. **M-AEGIS** achieves the three design goals by operating at a conceptual layer we call *Layer 7.5 (L-7.5)* that is positioned above the existing application layer (OSI Layer 7 [82]), and interacts directly with the user (popularly labeled as Layer 8 [92, 53]).

From a system's perspective, L-7.5 is a transparent window in an isolated process that interposes itself between the Layer 7 and 8. The interconnectivity among these layers is achieved using the accessibility framework, which is available as an essential feature of modern operating systems. Note that utilizing accessibility features for unorthodox purposes have been proposed by prior works [129, 85] that achieve different goals. L-7.5 extracts the GUI information of an app below it through the OS's user interface automation/accessibility (UIA) library. Using this information, **M-AEGIS** is then able to proxy user input by rendering its own GUI (with a different color as a visual cue) and subsequently handle those input (e.g., to process plaintext data or intercept user button click). Using the same UIA library, L-7.5 can also programmatically interact with various UI components of the app below on behalf of the user (refer to Figure 3.3.3 for more details). Since major software vendors today have pledged their commitment towards continuous support and enhancement of accessibility interface for developers [117, 9, 68, 5], our UIA-based technique is applicable and sustainable on all major platforms.

From a security design perspective, **M-AEGIS** provides two privacy guarantees during a user's interaction with a target app: 1) all input from the user first goes to L-7.5 (and is optionally processed) before being passed to an app. This means that confidential data and user intent can be fully captured; and 2) all output from the app must go through L-7.5 (and is optionally processed) before being displayed to the user.

From a developer's perspective, accessing and interacting with a target app's UI components at L-7.5 is similar to that of manipulating the Document Object Model (DOM) tree of a web app using JavaScript. While the DOM tree manipulation only works for browsers, UIA works for all apps on a platform. To track the GUI of an app, **M-AEGIS** relies on

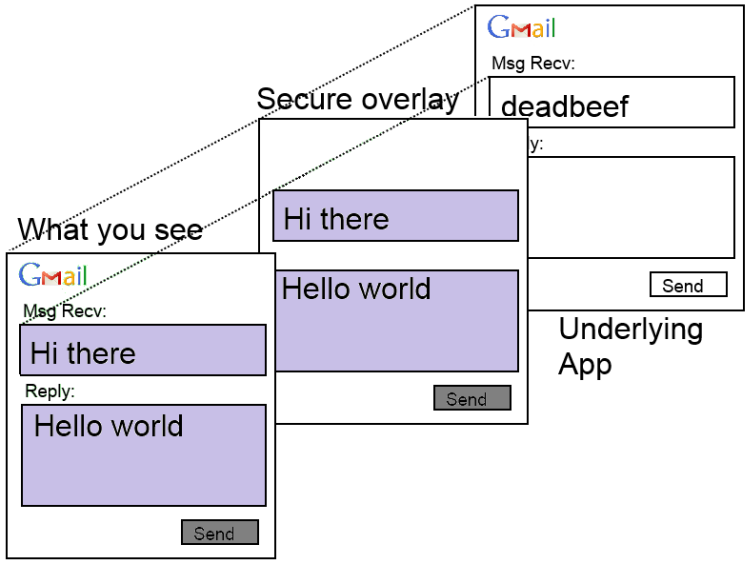

Figure 7: This diagram shows how M-Aegis uses L-7.5 to transparently reverse-transform the message "deadbeef" into "Hi there", and also allows a user to enter their plaintext message "Hello world" into M-Aegis's text box. To the user, the GUI looks exactly the same as the original app. When the user decides to send a message, the "Hello world" message will be transformed and relayed to the underlying app.

resource ID names available through the UIA library. Therefore, **M-AEGIS** is resilient to updates that change the look and feel of the app (e.g. GUI position or color). It only requires resource id names to remain the same, which, through empirical evidence, often holds true. Even if a resource id changes, minimal effort is required to rediscover resource id names and remap them to the logic in **M-AEGIS**. From our experience, **M-AEGIS** does not require developer attention across minor app updates.

From a user's perspective, **M-AEGIS** is visible as an always-on-top button. When it is turned on, users will perceive that they are interacting with the original app in plaintext mode. The only difference is the GUI of the original app will appear in a different color to indicate that protection is activated. This means that **M-AEGIS** preserves subtle features that contribute towards the entire user experience such as spell checking and in-app navigation. However, despite user perception, the original app *never* receives plaintext data. Figure 7 gives a high-level idea of how **M-AEGIS** creates an L-7.5 to protect user's data privacy when interacting with Gmail.

For users who would like to protect their email communications, they will also be concerned if encryption will affect their ability to search, as it is an important aspect of user productivity [159]. For this purpose, we designed and incorporated a new searchable encryption scheme named *easily-deployable efficiently-searchable symmetric encryption scheme* (EDESE) into **M-AEGIS** that allows search over encrypted content without any server-side modification. We briefly discuss the design considerations and security concerns involved in supporting this functionality in Figure 3.3.3.

As a proof of concept, we implemented a prototype **M-AEGIS** on Android that protects user data when interfacing with text-based PCS. **M-AEGIS** supports email apps like Gmail and messenger apps like Google Hangout, WhatsApp, and Facebook Chat. It protects data privacy by implementing E2EE that passes no plaintext to an app while also preserving the user experience and workflow. We also implemented a version of **M-AEGIS** on the desktop to demonstrate the generality of our approach. Our initial performance evaluation and user study show that users incur minimal overhead in adopting **M-AEGIS** on Android. There is imperceptible encryption/decryption latency and a low and adjustable false positive rate when searching over encrypted data.

In summary, these are the major contributions of this chapter:

- We introduced Layer 7.5 (L-7.5), a conceptual layer that directly interacts with users on top of existing apps. This is a novel system approach that provides seemingly contrasting features: transparent interaction with a target app and strong isolation from the target app.
- We designed and built **M-AEGIS** based on the concept of L-7.5, a system that preserves user privacy when interacting with PCS by ensuring data confidentiality. Essential functionalities of existing apps, especially search (even over encrypted data), are also supported without any server-side modification.
- We implemented two prototypes of **M-AEGIS**, one on Android and the other on Windows, with support for various popular public cloud services, including Gmail,

Facebook Messenger, Google Hangout, WhatsApp, and Viber.

- We designed and conducted a user study that demonstrated the acceptability of our solution.

3.2 Related Work

Since **M-AEGIS** is designed to achieve the three design goals described in §3.1 while seamlessly integrating end-to-end encryption into user's communication, we discuss how well existing works achieve some of these goals and ow they differ from **M-AEGIS**. As far as we know, no existing work achieves all the three design goals.

Standalone Solutions. There are many standalone solutions that aim to protect user data confidentiality. olutions like PGP [148] (including S/MIME [47]), Gibberbot [61], TextSecure [126], SafeSlinger [52], and FlyByNight [106] provide secure messaging and/or file transfer through encryption of user data. These solutions provide good isolation from untrusted entities. However, since they are designed as standalone custom apps, they do not preserve the user experience and require users to adopt a new workflow on a custom app. More importantly, these solutions are not retrofittable to existing apps on the mobile platform.

Like **M-AEGIS**, Cryptons [45] introduced a similarly strong notion of isolation through its custom abstractions. However, Cryptons assumes a completely different threat model that trusts PCS, and requires both server and client (app) modifications. Thus, Cryptons could not protect a user's communication using existing messaging apps while assuming the provider to be untrusted. We also argue that it is non-trivial to modify Cryptons to achieve the three design goals that we mentioned in §3.1.

Browser Plugin/Extension Solutions. Other solutions that focus on protecting user privacy include Cryptocat [98], Scramble! [17], TrustSplit [51], NOYB (None of Your Business) [71], and SafeButton [100]. Some of these assume different threat models and achieve different goals. For example, NOYB protects a user's Facebook profile data while

SafeButton tries to keep a user's browsing history private. Most of these solutions try to be transparently integrated into user workflow. However, since these solutions are mostly based on browser plugins/extensions, they are not applicable to the mobile platform.

Additionally, Cryptocat and TrustSplit require new and independent service providers to support their functionalities. However, **M-AEGIS** works with the existing service providers without assuming trust or requiring modification to server-side communication.

Repackaging/Rewriting Solutions. There is a category of work that repackages/rewrites an app's binary to introduce security features, such as Aurasium [166], Dr. Android [87], and others [20]. Our solution is similar to these approaches in that we can retrofit our solutions to existing apps and still preserve user experience, but is different in that **M-AEGIS**' coverage is not limited to apps that do not contain native code. Additionally, repackaging-based approaches suffer from the problem that they will break app updates. In some cases, attackers can circumvent the security of such solutions because the isolation model is unclear, i.e., the untrusted code resides in the same address space as the reference monitor (e.g., Aurasium).

Orthogonal Work. Although our work focuses on user interaction on mobile platforms with cloud providers, we assume a very different threat model than those that focus on more robust permission model infrastructures and those that focus on controlling/tracking information flow, such as TaintDroid [49] and Airbag [163]. These solutions require hanges to the underlying app, framework, or the OS, but **M-AEGIS** does not.

Access Control Gadgets (ACG) [133] uses user input as permission granting intent to allow apps to access user owned resources. Although we made the same assumptions as ACG to capture authentic user input, the design of ACG aims to provide a different threat model and security goal than ours. Furthermore, ACG requires a modified kernel but **M-AEGIS** does not. Persona [13] presents a completely isolated and new online social network that provides certain privacy and security guarantees to the users. While related, it differs from the goal of **M-AEGIS**. Frientegrity [55] and Gyrus (in §2) focus on different aspects of

integrity protection of a user's data. Tor [44] is well known for its capability to hide a user's IP address while browsing the Internet. However, it focuses on anonymity guarantees while **M-AEGIS** focuses on data confidentiality guarantees. Off-the-record messaging (OTR) [25] is a secure communication protocol that provides perfect forward secrecy and malleable encryption. While OTR can be implemented on **M-AEGIS** using the same design architecture to provide these extra properties, it is currently not the focus of our work.

3.3 System Design

3.3.1 Design Goals

In this section, we formally reiterate our design goals. We posit that a good solution must:

- Offer good security by applying strong isolation from untrusted entities (defined in §3.3.2).
- Preserve the user experience by providing users transparent interaction with existing apps.
- Be easy to maintain and scale by devising a sufficiently general-purpose approach.

Above all, these goals must be satisfied within the unique set of constraints found in the mobile platform, including user experience, transparency, deployability, and adoptability factors.

3.3.2 Threat Model

In-Scope Threats. We begin with the scope of threats that **M-AEGIS** is designed to protect against. In general, there are three parties that pose threats to the confidentiality of users' data exposed to public cloud through mobile devices. Therefore, we assume these parties to be untrusted in our threat model:

- Public cloud service (PCS) providers. Sensitive data stored in the public cloud can be compromised in several ways:
    1. PCS providers can be compelled by law [1] to provide access to a user's sensitive

data to law enforcement agencies [28];

2. The business model of PCS providers creates a strong incentive for them to share/sell user data to third parties [15, 2, 17, 51];
3. PCS administrators who have access to the sensitive data may also compromise the data, either intentionally [28] or not [43]; and
4. Vulnerabilities of the PCS can be exploited by attackers to exfiltrate sensitive data [74].

- Client-side apps. Since client-side apps are developed by PCS providers to allow a user to access their services, we consider these apps as untrusted.
- Middle boxes between a PCS and a client-side app. Sensitive data can also be compromised when it is transferred between a PCS and a client-side app. Incorrect protocol design/implementation may allow attackers to eavesdrop on plaintext data or perform Man-in-the-Middle attacks [50, 14, 40].

M-AEGIS addresses the above threats by creating L-7.5, which it uses to provide end-to-end encryption (E2EE) for user private data. We consider the following components as our trusted computing base (TCB): the hardware, the operating system (OS), and the framework that controls and mediates access to hardware. In the absence of physical input devices (e.g., mouse and keyboard) on mobile devices, we additionally trust the soft keyboard not to leak the keystrokes of a user. We rely on the TCB to correctly handle I/O for **M-AEGIS** and to provide proper isolation between **M-AEGIS** and untrusted components.

Additionally, we also assume that all the components of **M-AEGIS**, including L-7.5 that it creates, are trusted. The user is also considered trustworthy under our threat model in his intent. This means that he is trusted to turn on **M-AEGIS** when he wants to protect the privacy of his data during his interaction with the PCS.

Out of Scope Threats. Our threat model does not consider the following types of attacks. First, **M-AEGIS** only guarantees the confidentiality of a user's data, but not its availability.

Therefore, attacks that deny access to data (denial-of-service) either at the server or the client are beyond the scope of this work. Second, any attacks against our TCB are orthogonal to this work. Such attacks include malicious hardware [95], attacks against the hardware [161], the OS [88], the platform [154] and privilege escalation attacks (e.g., unauthorized rooting of a device). However, note that **M-AEGIS** can be implemented on a design that anchors its trust on trusted hardware and hypervisor (e.g. Gyrus [85] and Storage Capsules [24]) to minimize the attack surface against the TCB. Third, **M-AEGIS** is designed to prevent any direct flow of information from an authorized user to untrusted entities. Hence, leakages through all side-channels [153] are beyond the scope of this work.

Since the user is assumed to be trustworthy under our threat model to use **M-AEGIS** correctly, **M-AEGIS** does not protect the user against social-engineering-based attacks. For example, phishing attacks to trick users into either turning off **M-AEGIS** and/or entering sensitive information intounprotected UI components are beyond the scope of this work. Instead, **M-AEGIS** deploys best-effort protection by coloring the UI components in L-7.5 differently from that of the default app UI.

The other limitations of **M-AEGIS**, which are not security threats, are discussed in §3.6.2.

3.3.3 M-Aegis Architecture

M-AEGIS is architected to fulfill all of the three design goals mentioned in §3.3.1. Providing strong isolation guarantees is first. To achieve this, **M-AEGIS** is designed to execute in a separate process, though it resides in the same OS as the target client app (TCA). Besides memory isolation, the filesystem of **M-AEGIS** is also shielded from other apps by OS app sandbox protection.

Should a greater degree of isolation be desirable, an underlying virtual-machine-based system can be adopted to provide even stronger security guarantees. However, we do not consider such design at this time as it is currently unsuitable for mobile platforms, and the

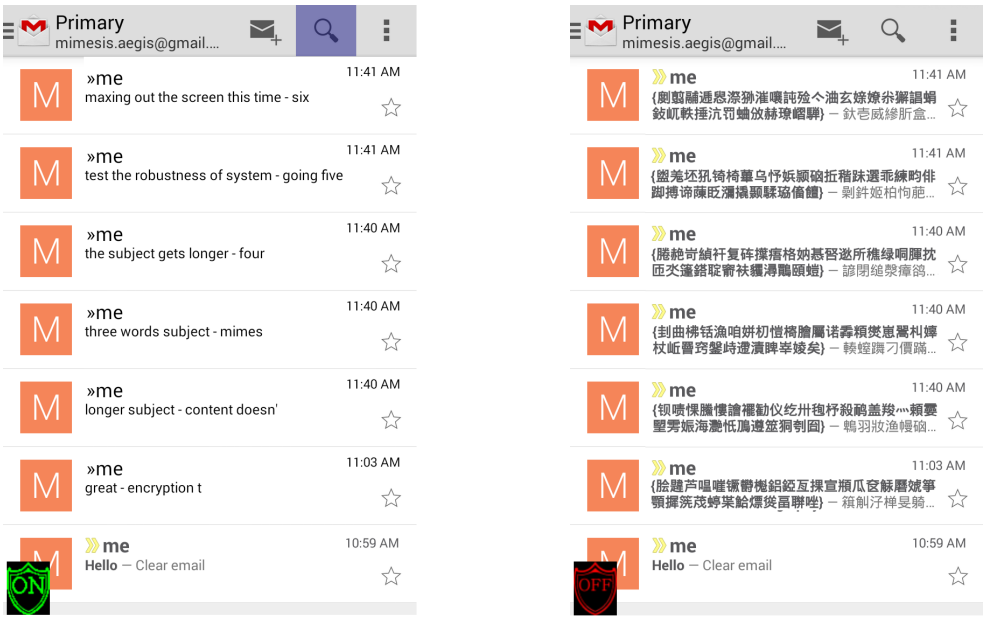

Figure 8: The figure on the left illustrates how a user perceives the Gmail preview page when M-Aegis is turned on. The figure on the right illustrates the same scenario but with M-Aegis turned off. Note that the search button is painted with a different color when M-Aegis is turned on.

adoption of such technology is beyond the scope of this work.

The main components that make up **M-AEGIS** are as follows.

Layer 7.5 (L-7.5). **M-AEGIS** creates a novel and conceptual layer called Layer 7.5 (L-7.5) to interpose itself between the user and the TCA. This layer allows **M-AEGIS** to implement a true end-to-end encryption (E2EE) without exposing plaintext data to the TCA while maintaining the TCA's original functionalities and user experience, fulfilling the second design goal. L-7.5 is built by creating a transparent window that is always-on-top. This technique is advantageous in that it provides a natural way to handle user interaction, thus preserving user experience without the need to reverse engineer the logic of TCAs or the network protocols used by the TCAs to communicate with their respective cloud service backends, fulfilling the third design goal.

There are three cases of user interactions to handle. The first case considers interactions that do not involve data confidentiality (e.g., deleting or relabeling email). Such input does not require extra processing/transformation and can be directly delivered to the underlying

TCA. Such click-through behavior is a natural property of transparent windows and helps **M-AEGIS** maintain the look and feel of the TCA.

The second case considers interactions that involve data confidentiality (e.g., entering messages or searching encrypted email). Such input requires extra processing (e.g., encryption and encoding operations). For such cases, **M-AEGIS** places opaque GUIs that "mimic" the GUIs over the TCA, which are purposely painted in different colors for two reasons: 1) as a placeholder for user input so that it does not leak to the TCA, and 2) for user visual feedback. Mimic GUIs for the subject and content as seen in Figure 9 are examples of this case. Since L-7.5 is always on top, this provides the guarantee that user input always goes to a mimic GUI instead of the TCA.

The third case considers interactions with control GUIs (e.g. send buttons). Such input requires user action to be "buffered" while the input from the second case is being processed before being relayed to the actual control GUI of the TCA. For such cases, **M-AEGIS** creates semi-transparent mimic GUIs that register themselves to absorb/handle user clicks/taps. Again, these mimic GUIs are painted with a different color to provide a visual cue to a user. Examples of these include the purple search button in the left figure in Figure 8 and the purple send button in Figure 9. Note that our concept of intercepting user input is similar to that of ACG's [133] in capturing user intent, but our application of the user intent differs.

UIA Manager (UIAM). To be fully functional, **M-AEGIS** requires certain capabilities that are not available to regular apps. First, although **M-AEGIS** is confined within the OS' app sandbox, it must be able to determine with which TCA the user is currently interacting. This information allows **M-AEGIS** to invoke specific logic to handle the TCA, and helps **M-AEGIS** clean up the screen when the TCA is terminated. Second, **M-AEGIS** requires information about the GUI layout for the TCA it is currently handling. This information allows **M-AEGIS** to accurately render mimic GUIs on L-7.5 to intercept user I/O. Third, although isolated from the TCA, **M-AEGIS** must be able to communicate with the TCA to maintain functionality and ensure user experience is not disrupted. For example, **M-AEGIS**

must be able to relay user clicks to the TCA, eventually send encrypted data to the TCA, and click on TCA's button on behalf of the user. For output on the screen, it must be able to capture ciphertext so that it can decrypt it and then render it on L-7.5.

M-AEGIS extracts certain features from the underlying OS's accessibility framework, which are exposed to developers in the form of User Interface Accessibility/Automation (UIA) library. Using UIA, **M-AEGIS** is not only able to know which TCA is currently executing, but it can also query the GUI tree of the TCA to get detailed information about how the page is laid out (e.g., location, size, type, and resource-id of the GUI components). More importantly, it can obtain information about the content of these GUI items.

Exploiting UIA is advantageous to our design as compared to other methods of information capture from the GUI, e.g., optical character recognition (OCR). Besides having perfect content accuracy, our technique is not limited by screen size. For example, even though the screen size may prevent full text to be displayed, **M-AEGIS** is still able to capture text in its *entirety* through the UIA libraries, allowing us to apply decryption to ciphertext comfortably. We thus utilize all these capabilities and advantages to build a crucial component of **M-AEGIS** called the UIA manager (UIAM).

Per-TCA Logic. **M-AEGIS** can be extended to support many TCAs. For each TCA of interest, we build per-TCA logic as an extension module. The per-TCA logic is responsible for rendering the specific mimic GUIs according to information it queries from the UIAM. Therefore, per-TCA logic is responsible for handling direct user input. Specifically, it decides whether the user input will be directly passed to the TCA or be encrypted and encoded before doing so. This decision ensures that the TCA never obtains plaintext data while user interaction is in plaintext mode. Per-TCA logic also intercepts button clicks so that it can then instruct UIAM to emulate the user's action on the button in the underlying TCA. Per-TCA logic also decides which encryption and encoding scheme to use according to the type of TCA it is handling. For example, encryption and encoding schemes for handling email apps would differ from that of messenger apps.

Cryptographic Module. M-AEGIS' cryptographic module is responsible for providing encryption/decryption and cryptographic hash capabilities to support our searchable encryption scheme (described in detail later) to the per-TCA logic so that M-AEGIS can transform/obfuscate messages through E2EE operations. Besides standard cryptographic primitives, this module also includes a searchable encryption scheme to support search over encrypted email that works *without server modification*. Since the discussion of any encryption scheme is not complete without encryption keys, the Key Manager is also a part of this module.

Key Manager. M-AEGIS has a key manager per TCA that manages key policies that can be specific to each TCA according to user preference. The key manager supports a range of schemes, including simple password-based key derivation functions (of which we assume the password to be shared out of band) to derive symmetric keys, which we currently implement as default, to more sophisticated PKI-based scheme for users who prefer stronger security guarantees and do not mind the additional key set-up and exchange overheads. However, the discussion about the best key management/distribution policy is beyond the scope of this work.

Searchable Encryption Scheme (EDESE). There are numerous encryption schemes that support keyword search [65, 146, 64, 29, 39, 23, 93]. These schemes exhibit different tradeoffs between security, functionality, and efficiency, but *all* of them require modifications on the server side. Schemes that make use of inverted index [39] are not suitable, as updates to inverted index cannot be practically deployed in our scenario.

Since we cannot assume server cooperation (consistent with our threat model in §3.3.2), we designed a new searchable encryption scheme called easily-deployable efficiently-searchable symmetric encryption scheme (EDESE). EDESE is an adaptation of a scheme proposed by Bellare et al. [19], with modifications similar to that of Goh's scheme [64] that is retrofittable to a non-modifying server scenario.

We incorporated EDESE for email applications with the following construct. The

idea for the construction is simple: we encrypt the document with a standard encryption scheme and append HMACs of unique keywords in the document. We discuss the specific instantiations of encryption and HMAC schemes that we use in §3.4.1. To prevent leaking the number of unique keywords we add as many dummy keywords as needed. We present this construction in detail in the full version of this work [102].

In order to achieve higher storage and search efficiency, we utilized a Bloom filter (BF) to represent the EDESE-index. In essence, a BF is a data structure that allows for efficient set-inclusion tests. However, such set-inclusion tests based on BFs are currently not supported by existing email providers, which only support string-based searches. Therefore, we devised a solution that encodes the positions of on-bits in a BF as Unicode strings (refer to §3.4.4 for details).

Since the underlying data structure that is used to support EDESE is a BF, search operations are susceptible to false positives matches. However, this does not pose a real problem to users, because the false positive rate is extremely low and is completely adjustable. Our current implementation follows these parameters: the length of keyword (in bits) is estimated to be $k = 128$, the size of the BF array is $B = 2^{24}$, the maximum number of unique keywords used in any email thread is estimated to be $d = 10^6$, the number of bits set to 1 for one keyword is $r = 10$. Plugging in these values into the formula for false positive calculation [64], i.e., $(1 - e^{-rd/B})^r$, we cap the probability of a false positive δ to 0.0003.

We formally assess the security guarantees that our construction provides. In the full version of this work [102], we propose a security definition for EDESE schemes and discuss why the existing notions are not suitable. Our definition considers an attacker who can obtain examples of encrypted documents of its choice and the results of queries of keywords of its choice. Given such an adversary, an EDESE scheme secure under our definition should hide all partial information about the messages except for the message length and the number of common keywords between any set of messages. Leaking the latter is unavoidable given that for the search function to be transparent to encryption, the output of a query has to be a

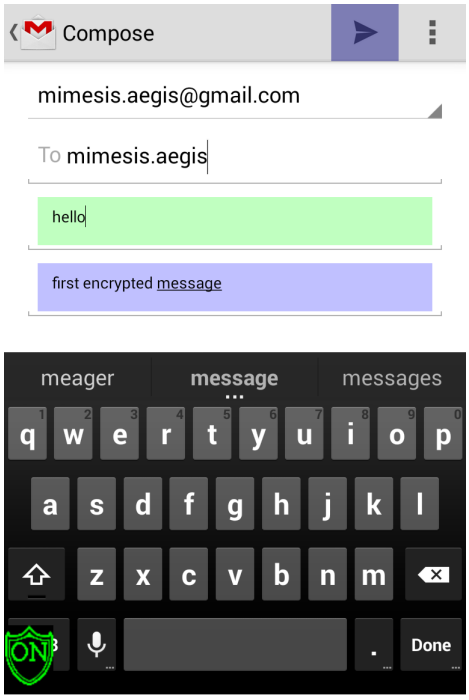

Figure 9: User still interacts with Gmail app to compose email, with M-Aegis' mimic GUIs painted with different colors on L-7.5.

part a ciphertext. But everything else, e.g., the number of unique keywords in a message, positions of the keywords, is hidden.

Given the security definition in the full version of this work [102], we prove that our construction satisfies it under the standard notions of security for encryption and HMACs.

3.3.4 User Workflow

To better illustrate how the different components in **M-AEGIS** fit together, we describe an example workflow of a user composing and sending an email using the stock Gmail app on Android using **M-AEGIS**:

1. When the user launches the Gmail app, the UIAM notifies the correct per-TCA logic of the event. The per-TCA logic will then initialize itself to handle the Gmail workflow.

2. As soon as the Gmail app is launched, the per-TCA logic will try to detect the state of the Gmail app (e.g., preview, reading, or composing email). This detection allows **M-AEGIS** to create mimic GUIs on L-7.5 to handle user interaction properly. For

example, when a user is on the compose page, the per-TCA logic will mimic the GUIs of the subject and content fields (as seen in Figure 9). The user then interacts directly with these mimic GUIs in plaintext mode without extra effort. Thus, the workflow is not affected at all. Note that essential but subtle features like spell check and autocorrect are still preserved, as they are innate features of the mobile device's soft keyboard. Additionally, the "send" button is also mimicked to capture user intent.

3. As the user finishes composing his email, he clicks on the mimicked "send" button on L-7.5. Since L-7.5 receives the user input and not the underlying Gmail app, the per-TCA logic can capture this event and proceed to process the subject and the content.

4. The per-TCA logic selects the appropriate encryption key to be used based on the recipient list and the predetermined key policy for Gmail. If a key cannot be found for this conversation, **M-AEGIS** prompts the user (see Figure 10) for a password to derive a new key. After obtaining the associated key for this conversation, **M-AEGIS** will then encrypt these inputs and encode it back to text such that Gmail can consume it.

5. The per-TCA logic then requests the UIAM to fill in the corresponding GUIs on Gmail with the transformed text. After they are filled, the UIAM is instructed to click the actual "send" button on behalf of the user. This procedure provides a transparent experience to the user.

This workflow evidently shows that because of the mimicking properties of **M-AEGIS**, the workflow of using Gmail remains the same from the user's perspective.

3.4 Implementation and Deployment

In this section, we discuss important details of our prototype implementations. We implemented a prototype of **M-AEGIS** using Java on Android, as an accessibility service. This is done by creating a class that extends the `AccessibilityService` class and requesting the

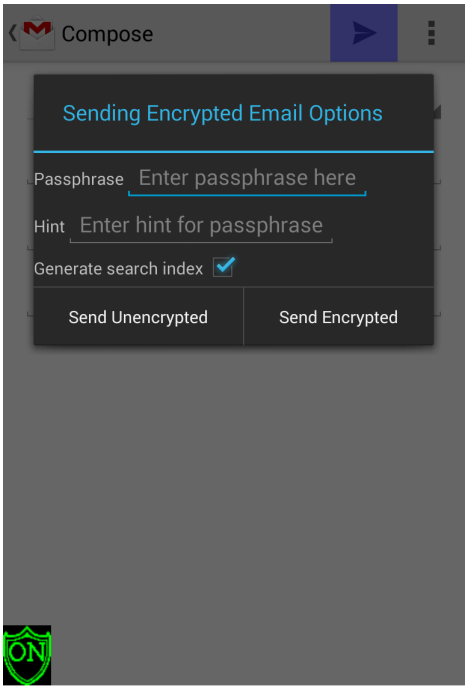

Figure 10: Password prompt when user sends encrypted mail for a new conversation.

`BIND_ACCESSIBILITY_SERVICE` permission in the manifest. This allows us to interface with the UIA library, building our UIAM. We discuss this in further detail in §3.4.2.

We then deployed our prototype on two Android phones from separate manufacturers, namely Samsung Galaxy Nexus and LG Nexus 4, targeting several versions of Android, from Android 4.2.2 (API level 17) to Android 4.4.2 (API level 19). The deployment was done on stock devices and OSes, i.e., *without* modifying the OS, Android framework, or rooting. Only simple app installation was performed. This demonstrates the ease of deployment and distribution of our solution. We have also implemented an **M-AEGIS** prototype on Windows 7 to demonstrate interoperability and generality of approach, but we do not discuss the details here, as it is not the focus of this work.

As an interface to the user, we create a button that is always on top even if other apps are running on the screen. This overlaying allows us to create a non-bypassable direct channel of communication with the user besides providing a visual cue of whether **M-AEGIS** is turned on or off.

For app support, we use Gmail as an example of an email app and WhatsApp as an

example of a messenger app. We argue that it is easy to extend the support to other apps within these classes.

We first describe the cryptographic schemes that we deployed in our prototype, then explain how we build our UIAM and create L-7.5 on Android, and finally discuss the per-TCA logic required to support both classes of apps.

3.4.1 Cryptographic Schemes

For all the encryption/decryption operations, we use `AES-GCM-256`. For a password-based key generation algorithm, we utilized ccPBKDF2 with `SHA-1` as the keyed-hash message authentication code (HMAC). We also utilized `HMAC-SHA-256` as our HMAC to generate tags for email messages (§3.4.4). These functionalities are available in Java's `javax.crypto` and `java.security` packages.

For the sake of usability, we implemented a password-based scheme as the default, and we assume one password for each group of message recipients. We rely on the users to communicate the password to the receiving parties using out of band channel (e.g. in person or phone calls). For messaging apps, we implemented an authenticated Diffie-Hellman key exchange protocol to negotiate session keys for WhatsApp conversations. A PGP key is automatically generated for a user during installation based on the hashed phone number, and is deposited in publicly accessible repositories on the user's behalf (e.g., MIT PGP Key Server [121]). Further discussion about verifying the authenticity of public keys retrieved from such servers is omitted from this chapter. Since all session and private keys are stored locally for user convenience, we make sure that they are never saved to disk in plaintext. They are additionally encrypted with a key derived from a master password that is provided by the user during installation.

3.4.2 UIAM

As mentioned earlier, UIAM is implemented using UIA libraries. On Android, events that signify something new being displayed on the screen can be detected by monitoring following the events: `WINDOW_CONTENT_CHANGED`, `WINDOW_STATE_CHANGED`, and `VIEW_SCROLLED`. Upon receiving these events, per-TCA logic is informed by UIAM. The UIA library presents a data structure in the form of a tree with nodes representing UI components with the root being the top window. This tree structure allows UIAM to locate all UI components on the screen.

Additionally, Android's UIA framework also provides the ability to query for UI nodes by providing a resource ID. For instance, we can find the node that represents Gmails search button by querying for `com.google.android.gm:id/search`. More importantly, there is no need to guess the names of these resource IDs. Rather, we use a tool called UI Automator Viewer [7] (see §3.4.4), which comes with the default Android SDK. Once we found the node of interest, we can get all the other information about the GUI represented by the node. This information includes the exact location and size of text boxes and buttons on the screen. **M-AEGIS** can programmatically interact with various GUIs of a TCA using the function `performAction()`. This function allows it to click on a TCAs button on the users behalf after it has processed the user input.

3.4.3 Layer 7.5

We implemented Layer 7.5 on Android as specific types of system windows, which are *always-on-top* of all other running apps. Android allows the creation of various types of system windows. We focus on two, `TYPE_SYSTEM_OVERLAY` and `TYPE_SYSTEM_ERROR`. The first is for displaying only and allowing all tap/keyboard events to go to underlying apps. In contrast, the second type allows for user interaction. Android allows the use of any `View` objects for either type of window, and we use this to create our mimic GUIs, and set their size and location. We deliberately create our mimic GUIs in different colors as a subtle

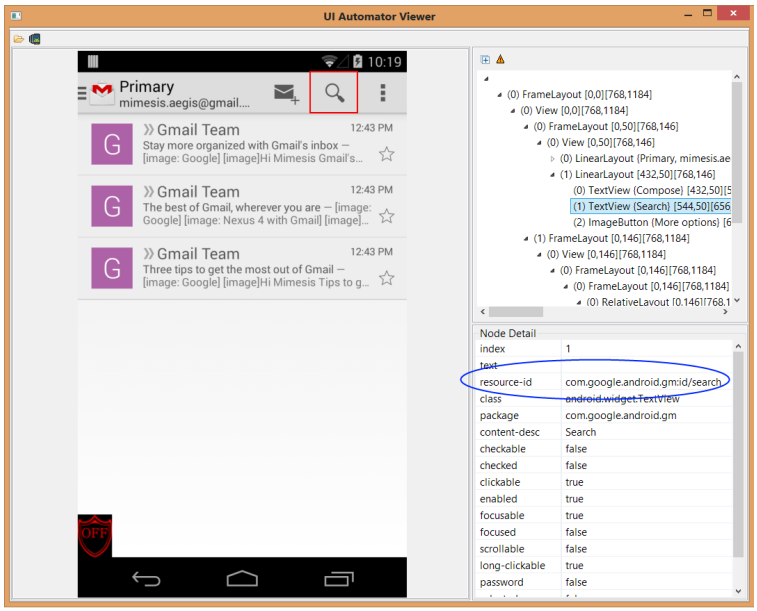

Figure 11: The UI Automator Viewer presents an easy to use interface to examine the UIA tree and determine the resource ID (blue ellipse) associated with a GUI of interest (red rectangle)

visual cue to the users that they are interacting with **M-AEGIS**, without distracting them from their original workflow.

3.4.4 Per-TCA Logic

From our experience developing per-TCA logic, the general procedure for development is as follows:

- Understand what the app does. First, we need to identify which GUIs need to be mimicked by the TCA logic for intercepting user I/O. For text-based TCAs, this is a trivial step because the core functionalities that **M-AEGIS** needs to handle are limited and thus easy to identify, e.g., buffering users typed texts and sending them to the intended recipient.
- Using UI Automator Viewer [7], examine the UIA tree for the relevant GUIs of a TCA and identify signatures (GUI resource IDs) for each TCA state. UI Automator Viewer allows inspection of the UIA tree through a graphical interface (as seen in Figure 11), which reduces development time. We rely on UI components that are unique to individual states (e.g., the "send" button signifies that we are in the compose

state).

- For each relevant GUI, we need to devise algorithms to extract either the location and content of ciphertext (for decryption and display), or the type, size, and location of GUIs we need to mimic (e.g., the subject and content boxes in the Gmail compose UI). Again, this is done through UI Automator Viewer. For example, for the Gmail preview state, we query the UIA for nodes with ID `com.google.android.gm:/id/conversation_list` to identify all the UIA nodes corresponding to the preview item of individual email, and from those, we can extract all ciphertext on the preview window through the UIA).
- Create event handlers for controls we mimic on L-7.5. For the Gmail compose state, we need to listen for click/touch events for the L-7.5 "send" button and carry out the process described in Figure 3.3.3 to encrypt the email and send the ciphertext to the underlying TCA.
- Identify ways that each relevant state can be updated. Updates can be handled via the following method: clear L-7.5, extract all necessary information from the new state, and then render again. This procedure is equivalent to redrawing all GUIs on L-7.5 based on the detected state.

There are two details worth considering when developing per-TCA logic. First, careful consideration must be given to the type of input data fed to TCAs. Since most TCAs only accept input data in specific formats, e.g., text, they do not support the input of random byte sequences as valid data. Therefore, encrypted data must be encoded into text format before feeding it as input to a TCA. Conventionally, `base64` encoding is used for this purpose. However, `base64` encoding consumes too much on-screen real estate. To overcome this, we encoded the binary encrypted data into Chinese Japanese Korean (CJK) Unicode characters, which have efficient on-screen real estate consumption. To map the binary data into the CJK plane, we process the encrypted data at the byte granularity (2^8). For each byte, its value is added to the base of the CJK Unicode representation, i.e., 0x4E00. For example, byte 0x00

will be encoded as '一', and byte 0x01 will be represented as '丅'.

Second, **M-AEGIS** can only correctly function if it can differentiate between ordinary messages and encrypted messages. We introduce markers into the encrypted data after encoding. In particular, we wrap the subject and content of a message using a pair of curly braces (i.e. {, }).

Next, we describe implementation details that are specific to these classes of apps. We begin by introducing the format of message we created for each class. Then we discuss other caveats (if any) that are involved in the implementation.

Email Apps. We implemented support for Gmail on our prototype as a representative app of this category. We create two custom formats to communicate the necessary metadata to support **M-AEGIS**' functionalities:

- Subject: $\{Encode(ID_{Key}||IV||Encrypt(Subject))\}$
- Content: $\{Encode(Encrypt(Content)||Tags)\}$

A particular challenge that we faced in supporting decryption during the Gmail preview state is that only the beginning parts of both the title and the subject of each message are available to us. Additionally, the exact email addresses of the sender and recipients are not always available, as some are displayed as aliases, and some are hidden due to lack of space. The lack of such information makes it impossible to automatically decrypt the message even if the corresponding encryption key actually exists on the system.

To solve these problems, when we encrypt a message, we include a key-ID (ID_{Key}) to the subject field (as seen in the format described above). Note that since the key-ID is not a secret, it need not be encrypted. This way, we will have all the information we need to decrypt the subtext displayed on the Gmail preview correctly.

The $Tags$ field is a collection of HMAC digests that are computed using the conversation key and keywords that exist in a particular email. It is then encoded and appended as part of the content that Gmail receives to facilitate encrypted search without requiring modification to Gmail's servers.

Messenger Apps. We implemented support for WhatsApp on our prototype as a representative app of this category. The format we created for this class of apps is simple, as seen below:

- Message: $\{Encode(IV||Encrypt(Message))\}$

We did not experience additional challenges when supporting WhatsApp.

3.5 Evaluations

In this section, we report the results of experiments to determine the correctness of our prototype implementation, measure the overheads of **M-AEGIS**, and user acceptability of our approach.

3.5.1 Correctness of Implementation

We manually verified **M-AEGIS**'s correctness by navigating through different states of the app and checking if **M-AEGIS** creates L-7.5 correctly. We manually verified that the encryption and decryption operations of **M-AEGIS** work correctly. We ensured that plaintext is properly received at the recipient's end when the correct password is supplied. We manually verified the correctness of our searchable encryption scheme by introducing specific search keywords. We performed the search using **M-AEGIS** and found no false negatives in the search result.

3.5.2 Performance on Android

The overhead that **M-AEGIS** introduced to a user's workflow can be broken down into two factors: 1) the additional computational costs incurred during encryption and decryption of data, and 2) the additional I/O operations when redrawing L-7.5. We measure overhead by measuring the overall latency presented to the user in various use cases. We found that **M-AEGIS** imposes negligible latency to the user.

We exercised all test cases on a *stock* Android phone (LG Nexus 4), with the following specifications: Quad-core 1.5 GHz Snapdragon S4 Pro CPU, equipped with 2.0 GB RAM,

running Android Kit Kat (4.4.2, API level 19). Unless otherwise stated, we repeated each experiment for ten times and took the averaged result for reporting.

For our evaluation, we only performed experiments for the setup of the Gmail app because Gmail is representative of a more sophisticated TCA, and thus indicates worst-case performance for **M-AEGIS**. Messenger apps incur fewer overheads given their simpler TCA logic.

Previewing Encrypted Email. There are additional costs involved in previewing encrypted emails on the main page of Gmail. The costs are broken down into times taken to 1) traverse the UIA tree to identify preview nodes, 2) capture ciphertext from the UIA node, 3) obtain the associated encryption key from the key manager, 4) decrypting ciphertext, and 5) rendering plaintext on L-7.5. We measure these operations as a single entity by running a macro benchmark.

For our experiment, we ensured that the preview page consists of encrypted emails (a total of six can fit on-screen) to demonstrate worst-case performance. We measured the time taken to perform all operations. We found, on average, it takes an additional 76 ms to render plaintext on L-7.5. Note that this latency is well within expected response time (50 - 150 ms), beyond which a user would notice the slowdown effect [145].

Composing and Sending Encrypted Email. We measured the extra time taken for encrypting a typical email and building our searchable encryption index for it. We used the Enron Email Dataset [34] as a representation of typical emails. We randomly picked ten emails. The average number of words in an email is 331, of which 153 are unique. The shortest sampled email contained 36 words, of which 35 are unique. The longest sampled email contains 953 words, of which 362 are unique.

With the longest sampled email, **M-AEGIS** took 205 ms in total to both encrypt and build the search index. Note that this includes the network latency a user will perceive while sending an email, regardless of their use of **M-AEGIS**.

Searching on Encrypted Emails. A user usually inputs one to three keywords per search

operation. The latency experienced when performing a search is negligible. This is because the transformation of the actual keyword into indexes requires only the forward computation of one HMAC, which is nearly instantaneous.

3.5.3 User Acceptability Study

This section describes the user study we performed to validate our hypothesis of user acceptability of **M-AEGIS**. Users were sampled from a population of college students. They must be able to operate smartphones proficiently and have had previous experience using the Gmail app. Each experiment was conducted with two identical smartphones, i.e., Nexus 4, both running Android 4.3, installed with the stock Gmail app (v. 4.6). One of the devices had **M-AEGIS** installed.

The setup of the experiment is as follows. We asked the user to perform a list of tasks: previewing, reading, composing, sending, and searching through email on a device that is not equipped with **M-AEGIS**. Participants were asked to pay attention to the overall experience of performing such tasks using the Gmail app. This experiment served as the control experiment.

Participants were then told to repeat the same set of tasks on another device that was equipped with **M-AEGIS**. This experiment was done with the intention that they were able to mentally compare the difference in user experience when interacting with the two devices.

We queried the participants if they found any difference in previewing, reading, sending, and searching email, and if they felt that their overall experience using the Gmail app on the second device was significantly different.

We debriefed the participants about the experiment process and explained the goal of **M-AEGIS**. We asked them whether they would use **M-AEGIS** to protect the privacy of their data. The results we collected and report here are from 15 participants.

We found that no participants noticed major differences between the two experiences using the Gmail app. One participant noticed a minor difference in the email preview

interface, i.e., L-7.5 did not catch up smoothly when scrolled. A different participant noticed a minor difference in the process of reading email, i.e., L-7.5 lag before covering ciphertext with mimic GUIs. There were only two participants that found the process of sending email differed from the original. When asked for details, they indicated that the cursor when composing email was not working properly. After further investigation, we determined this was a bug in Android's GUI framework rather than a fundamental flaw in **M-AEGIS**'s design.

Despite the perceived minor differences when performing particular tasks, all participants indicated that they would use **M-AEGIS** to protect the privacy of their data after understanding what **M-AEGIS** is. This feedback implies that they believe that the overall disturbance to the user experience is not large enough to impede adoption.

Since we recruited 15 users for this study, the accuracy/quality of our conclusion from this study lies between 80% and 95% (between 10 and 20 users) according to findings in [54]. We intend to continue our user study tto validate our acceptability hypothesis further and to continuously improve our prototype based on received feedback.

3.6 Discussions

3.6.1 Generality and Scalability

We believe that our **M-AEGIS** architecture presents a general solution that protects user data confidentiality, which is scalable in the following aspects:

Across Multiple Cloud Services. There are two main classes of apps that provide communication services, email and messenger apps. By providing functionality for apps in these two categories, we argue that **M-AEGIS** can satisfy a large portion of mobile security user needs. The different components of **M-AEGIS** incur a one-time development cost. We argue that it is easy to scale across multiple cloud services because the per-TCA logic that needs to be written is minimal per new TCA. This should be evident through the five general steps highlighted in §3.4.4. Additionally, the logic we developed for the first TCA (Gmail)

serves as a template/example to implement support for other apps.

Across App Updates. Since the robustness of the UIAM construct (§3.4.2) gives **M-AEGIS** the ability to track all TCA GUIs regardless of TCA state, **M-AEGIS** can easily survive app updates. Our Gmail app support has survived two updates without requiring major efforts to adapt.

Resource ID names can change across updates. For example, when upgrading to Gmail app version `4.7.2`, the resource ID name that identifies a sender's account name changed. Using UI Automator Viewer, we quickly discovered and modified the mapping in our TCA logic. Note that only the mapping was changed; the logic for the TCA does not need to be modified. This is because the core functionality of the updated GUI did not change (i.e., the GUI associated with a sender's account remained a text input box).

3.6.2 Limitations

As mentioned earlier, **M-AEGIS** is not designed to protect users against social-engineering-based attacks. Adversaries can trick users into entering sensitive information to the TCA while **M-AEGIS** is turned off. Our solution is best effort by providing distinguishing visual cues to the user when **M-AEGIS** is turned on, and its L-7.5 is active. For example, the mimic GUIs that **M-AEGIS** creates a different color. Users can toggle **M-AEGIS**' button on or off to see the difference (see Figure 8). Note that **M-AEGIS**'s main button is always on top and cannot be drawn over by other apps. However, we do not claim that this entirely mitigates the problem.

One of the constraints that we faced while retrofitting a security solution to existing TCAs (not limited to mobile environments) is that data must usually be of the right format (e.g., strictly text, image, audio, or video). For example, Gmail accepts only text (Unicode-compatible) for an email subject, but Dropbox accepts any type of files, including random blobs of bytes. Currently, other than the text format, we do not yet support other types of user data (e.g., image, audio, and video). However, this is *not* a fundamental design limitation of

our system. Rather, it is because of the unavailability of transformation functions (encryption and encoding schemes) that works for these media types.

Unlike text, the transformation/obfuscation functions in **M-AEGIS** for other types of data may also need to survive other process steps, such as compression. It is normal for TCAs to perform compression on multimedia to conserve bandwidth and/or storage. For example, Facebook is known to compress/downsample the image uploads.

The confidentiality guarantee that we provide excludes risks at the end points themselves. For example, a poor random number generator can potentially weaken the cryptographic schemes **M-AEGIS** applies. It is currently unclear how our text transformations will affect a server's effectiveness in performing spam filtering.

Our system currently does not tolerate typographical error during the search. However, we would like to point out that this is an unlikely scenario, given that soft keyboards on mobile devices utilize spell check and autocorrect features. Again, this is not a flaw in our architecture. Rather, it is because of the unavailability of encryption schemes that tolerate typographical error search without requiring server modification.

3.7 Summary

In this chapter, we presented *Mimesis Aegis (* **M-AEGIS** *)*, a new approach to protect private user data in public cloud services. **M-AEGIS** provides strong isolation and preserves user experience through the creation of a novel conceptual layer called *Layer 7.5 (L-7.5)*, which acts as a proxy between an app (Layer 7) and a user (Layer 8). This approach allows **M-AEGIS** to implement true end-to-end encryption of user data while achieving three goals:

1. Plaintext data is never visible to a client app, any intermediary entities, or the cloud provider;

2. the original user experience with the client app is preserved completely, from workflow to GUI look-and-feel; and

3. the architecture and technique are general to a large number of apps and resilient to app updates.

We implemented a prototype of M-Aegis on Android that can support a number of popular cloud services (e.g., Gmail, Google Hangout, Facebook, WhatsApp, and Viber). Our user study shows that our system preserves both the workflow and the GUI look-and-feel of the protected applications, and our performance evaluations show that users experienced minimal overhead in utilizing M-Aegis on Android. As the industry's follow-up after releasing the implementation detail of M-Aegis, many internet messenger company take the route to apply a true end-to-end (i.e., user-to-user) encryption on their messaging services. For example, in the year of 2016, WhatsApp applied a user-to-user, perfect forward secrecy encryption to its application and also released a whitepaper that describes their implementation for public verification. While we are hoping the other internet messenger vendors to adopt such approach, M-Aegis could be a soultion to apply before the vendors moving towards implementing more secure messengers.

CHAPTER IV

A11Y ATTACKS: EXPLOITING ACCESSIBILITY IN OPERATING SYSTEMS

4.1 Motivation

On August 9, 1998, the United States Congress amended the Rehabilitation Act of 1973 to eliminate barriers to electronic and information technology for people with disabilities [150]. Effective June 21, 2001, the law is enforced on the development, procurement, maintenance, or use of electronic and information technology by the federal government [114]. Driven by this requirement, OS vendors [117, 9, 69] have included accessibility features such as on-screen keyboards, screen magnifiers, voice commands, screen readers, etc. in their products to comply with federal law.

Assistive technologies, especially natural language voice processors, are gaining widespread market acceptance. Since the iPhone 4S, Apple has included in iOS a voice-based personal assistant, Siri, which can help the user complete a variety of tasks, such as placing a call, sending a text, and modifying personal calendars. Google also added a similar feature, Voice Action, to its Android platform. Furthermore, wearable devices such as Google Glass use voice as the primary interaction interface.

In general, adding new features into modern complex OSes usually introduces new security vulnerabilities. Accessibility support is no exception. For example, in 2007, it was reported that Windows Vista could be compromised through its speech recognition software [127]; in 2013, a flaw was discovered in Siri that allowed the bypass of an iPhone's lock screen to access photos and email [46]. As more and more people are using accessibility features, security issues caused by such vulnerabilities can become more serious.

In this work, we present the first security evaluation of the accessibility support of

commodity OSes. Our hypothesis is that alternative I/O subsystems such as assistive technologies bring a common challenge to many widely deployed security mechanisms in modern OSes. Modern OSes support restricted execution environments (e.g., sandboxes) and ask for the user's approval before applying a security sensitive change to the system (e.g., User Access Control (UAC) on Windows [118] and remote view on iOS [18]). However, accessibility support usually offers interfaces to programmatically generate user inputs, such as keystrokes and mouse clicks, which essentially enables the interface to act like a human being. Consequently, it might be pOSesible to bypass these defense mechanisms and abuse a user's permissions by generating synthesized user inputs. Similarly, attackers may also be able to steal security sensitive information displayed on the screen through the accessibility interfaces.

To verify our hypothesis, we examined the security of accessibility on four commodity OSes: Microsoft Windows 8.1, Ubuntu 13.10, iOS 6, and Android 4.4. We were able to identify *twelve* [1] attacks [2] that can bypass many state-of-the-art defense mechanisms deployed on these OSes, including UAC, the Yama security module, the iOS App sandbox, and the Android sandbox.

When designing new interfaces that provide access to computing systems, one must ensure that these new features do not break existing security mechanisms. However, current designs and implementations of accessibility support have failed to meet this requirement. Our analysis shows that current architectures for providing accessibility features make it extremely difficult to balance compatibility, usability, security, and (economic) cost. In particular, we found that security has received less consideration compared to the other factors. Under current architectures, there is not a single OS component that has all the information necessary to enforce meaningful security policy; instead, the security of accessibility features depends on security checks implemented in the assistive technology,

[1] We discovered eleven new attacks, and we cover an attack for Siri that was released in public as exploitation of accessibility in OS.

[2] **Disclosure:** we reported all vulnerabilities that we found to the OS vendors.

the OS, and the applications. Unfortunately, in our evaluation, we found that security checks are either entirely missed or implemented incorrectly (or incompletely) at all levels. Based on our findings, we believe a fundamental solution to the problem will involve a new architecture that is designed with security in mind. Proposing this new architecture is beyond the scope of our work. Instead, we propose several recommendations that work under current architectures to either make the implementation of all necessary security checks easier and more intuitive or to alleviate the impact of missing/incorrect checks. We also point out some open problems and challenges in automatically analyzing a11y support and identifying security vulnerabilities.

In summary, this chapter makes the following contributions:

- We performed a security evaluation of accessibility support for four major OSes: Windows, Ubuntu Linux, iOS, and Android;
- We found several new vulnerabilities that can be exploited to bypass many state-of-the-art defense mechanisms deployed on these systems, including UAC and application sandboxes;
- We analyzed the root cause of these vulnerabilities and proposed a number of recommendations to improve the security of a11y support;
- We showed that the current architectures for providing accessibility features are inherently flawed because no single OS component can implement a security policy: *security checks at the assistive technology, the OS, and the application must be implemented correctly; failure in any of these checks introduces vulnerabilities*.

4.2 Overview of Accessibility

In this section, we give a brief overview of accessibility in operating systems, and explain definitions of terminologies used in this work.

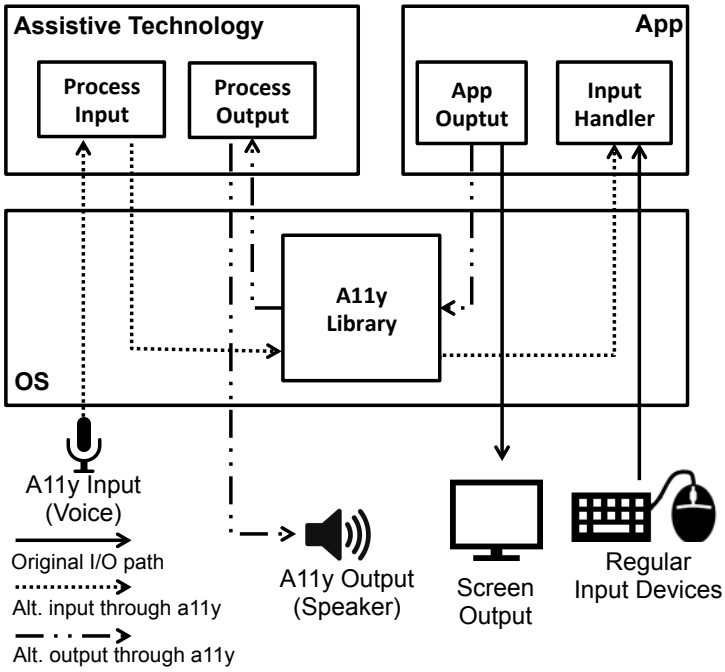

Figure 12: A general architecture for implementing accessibility features. Supporting an accessibility feature creates new paths for I/O on the system (two dotted lines), while original I/O from/to hardware devices (e.g., keyboard/mouse and screen) is indicated on the right side.

4.2.1 Accessibility Features

In compliance with the amended Rehabilitation Act, software vendors have incorporated various accessibility features into their systems. In this work, we define *computer accessibility (a11y) features* as new I/O subsystems that provide alternative ways for users with disabilities to interact with the system. For example, for visually impaired users, text-to-speech based Narrator (on MS Windows), VoiceOver (on OS X), and TalkBack (on Android) provide an output subsystem that communicates with the user through speech. For hearing impaired users, accessibility features like captioning services turn the system's audio output into visual output. Similarly, some systems can alert the user about the presence of audio output by flashing the screen. For users with motor disabilities, traditional mouse/keyboard based input systems are replaced by systems based on voice input. In general, we can see these accessibility features as implemented within an OS architecture in Figure 12.

There are also accessibility features that use traditional I/O devices (e.g., the screen,

mouse, and keyboard), but make them easier for users with disabilities to interact with the system. Examples of such features include:

- Magnifier in Windows, which enlarges certain portions of the screen;
- High Contrast in Windows, which provides higher contrast for easy distinction of user interfaces; and
- On-screen keyboard, sticky keys, filter keys, assisted pointing, and mouse double-click speed adjust to allow input requiring less movement.

4.2.2 Accessibility Libraries

In addition to pre-installed accessibility features, most OS vendors provide libraries for third parties to implement their own accessibility features. This makes it possible to create new alternative I/O subsystems based on other I/O devices (e.g., a braille terminal). In this case, the assistive technology part in Figure 12 is a program developed by the third party. Examples of these libraries include:

- UI Automation in Microsoft Windows,
- The accessibility toolkit (ATK) and Assistive Technology Service Provider Interface (AT-SPI) in Ubuntu Linux,
- AccessibilityService and related classes in Android, and
- The (public) NSAccessibility and (private) UIAutomation frameworks in iOS.

For all the discussions that follow, we will refer to these libraries as *accessibility libraries*. In general, the accessibility libraries provide the following capabilities as APIs:

- Notifications on changes to the system's display (e.g., new window popped up, content of a window changed/scrolled, change of focus, etc.);
- Ways to probe what is displayed on various UI elements (e.g., name of a button, content of a textbox, or static text displayed);
- Ways to synthesize inputs to various UI elements (e.g., click a button to place text into a textbox).

4.2.3 Assistive Technologies

For the rest of this chapter, we will use the term *assistive technology* (AT) to refer to the logic that runs in user space to provide any of the following functionality:

- (F1) processing user input from alternative input devices, "understanding" what the user wants and turning it into commands to the OS for control of other applications (or the OS itself);
- (F2) receiving information about the system's output and presenting it to users using alternative output devices.

Usually an assistive technology makes use of an accessibility library to obtain required capabilities for implementing a new accessibility feature.

4.3 Security Implications of A11y

In this section, we discuss new attack paths due to accessibility features in computing systems and correspondingly the required security checks for securing accessibility support. For the rest of this chapter, we adopt the threat model where the attacker controls one user space process with access to the accessibility library, and we do not assume any other special privilege for this malicious process.

4.3.1 New Attack Paths

The first functionality (F1) of AT allows users to control the system through alternative input devices, which is inherently dangerous from a security perspective. While modern OSes provide increasingly restricted isolation between applications, accessibility support provides a way to bypass this isolation and control other applications on the system.

To prevent malware from abusing security sensitive privileges of the user, OSes also deploy defense mechanisms such as User Account Control (UAC) [118] in Windows, remote view [18] in iOS, and ACG [133], with the policy of "ask for user consent explicitly before performing dangerous operations" (see Figure 13). However, since user-consent

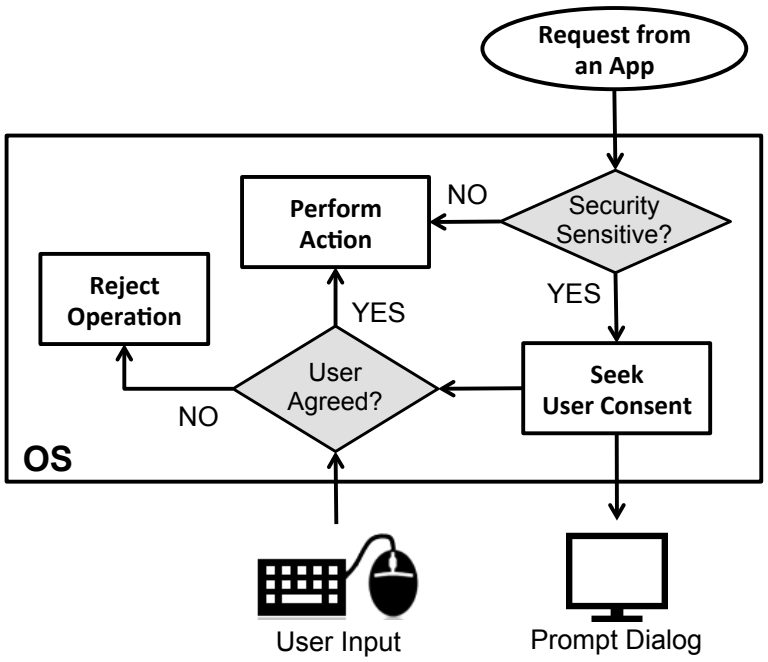

Figure 13: A workflow for the traditional mechanism to seek user consent before performing privileged operations.

is usually represented by a certain input event (e.g., click on a button), the capability to programmatically generate input events also breaks the underlying assumption of these security mechanisms that input is always the result of user action.

The ability of AT to monitor and probe the information currently being displayed on the screen (F2) is also problematic because it provides a way to access certain security sensitive information, e.g., plaintext passwords usually not displayed on the screen (e.g., most OSes show only scrambled symbols in the password box).

Based on the above observations, we argue that accessibility interfaces provide malware authors with these new paths of attacks:

- (A1) Malware implemented as AT penetrates the OS security boundary by obtaining new capabilities of controlling applications;
- (A2) Malware exploits the capability of generating interaction requests to bypass defense mechanisms or escalate its privilege;
- (A3) Malware exploits the capability of monitoring and probing the UI output to

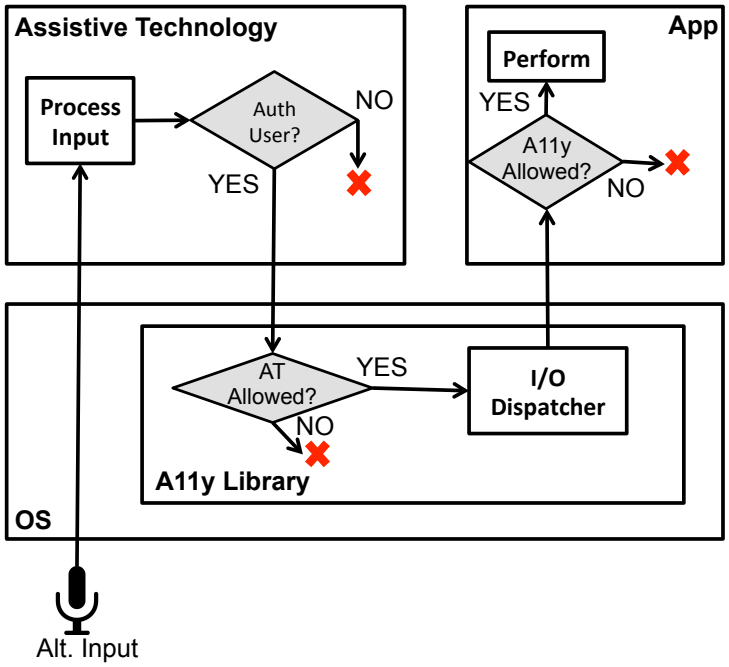

Figure 14: Required security checks for an AT as a new input subsystem. User input is passed to the AT first, moved to OS through accessibility libraries, then the synthetic input is delivered to the application. Grayed boxes indicate security checks required by each entity that receives the input.

access otherwise unavailable information.

4.3.2 Required Security Checks

To evaluate how a platform could be secure against these new attack paths, we propose two reference models of required checks: one for handling alternative input (Figure 14) and the other for handling output (Figure 15).

The key to securely handling alternative input is to validate whether the input is truly from the user. To achieve this goal, we argue that three checks (gray boxes in Figure 14) along the input path are necessary: within the AT, in the OS, and at the application level.

First, an AT should validate whether the input is from the user. Otherwise, attacks can be launched by synthesizing the input format of this AT. For example, malware can transform malicious operations into synthetic voice (e.g., via text-to-speech, TTS) and drive the natural language user interface to control other applications (A1) or escalate its privilege (A2).

Second, since not all ATs can be trusted (e.g., those provided by a third-party), the OS

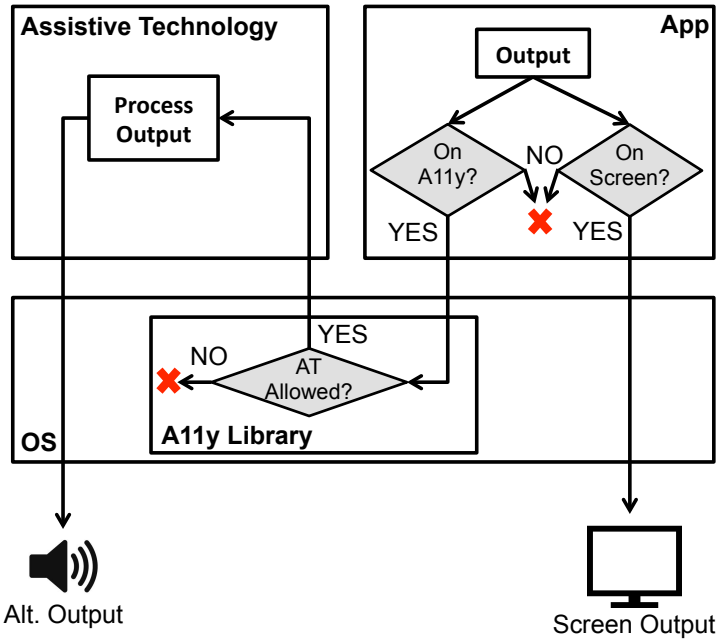

Figure 15: Required security checks for an AT as a new output subsystem. The application is required to decide which input can transit through the accessibility library. Then the AT receives the output to deliver it to the user. Grayed boxes indicate the checks required by OS and the application.

should have control over what applications an AT can control. For example, interaction requests from untrusted AT to security sensitive processes such as system services and system settings should not be forwarded. Otherwise, privilege escalation would be feasible (A2). Additionally, the access control policy should be consistent with other access control mechanisms to prevent a malicious AT from obtaining new capabilities (A1).

Third, the OS should provide the flexibility to allow an application to specify a fine-grained security policy on how to handle interaction requests from an AT. More specifically, the OS should 1) allow the application to distinguish input from real hardware and input from AT; and 2) allow the application to set its own callback functions to handle input events from AT. More importantly, when no customization is provided, the default setting should align with the platform's default security policy.

These three checks are complementary to each other for the following reasons. First, for AT-like natural language user interfaces for motor disabled people, it has to be able to control all applications and the underlying system; the only viable check is within the AT

itself. Second, as not all ATs are trustworthy, the OS-level check is necessary to prevent malicious AT from compromising the system. Third, OS-level access controls are not aware of the context of each non-system application, so the application level check provides the last line of defense for an application to protect itself from malicious ATs (A1).

Similarly, to securely handle alternative output and prevent information leakage (A3), two checks (gray boxes in Figure 15) should be performed. The application level check allows the application to specify what information is sensitive so it will not be available to AT. Again, we must emphasize that when no customization is provided, the default setting should align with the platform's security policy. The OS-level check prevents untrusted ATs from acquiring sensitive information specific to the system.

4.4 Security Evaluation of A11y

In this section, we first describe our evaluation methodology and then present the results of the security evaluation on major platforms: Microsoft Windows, Ubuntu Linux, iOS, and Android. The specific versions of the evaluated systems are: Windows 8.1, Ubuntu 13.10, iOS 6 and Android 4.4 on the Moto X [3].

4.4.1 Evaluation Methodology

Given an OS platform, we evaluate the security of the accessibility features it offers as follows:

- We studied the availability of the built-in assistive technologies and the accessibility library on the platform. For built-in assistive technologies, we focused on the availability of a natural language user interface because it provides the most powerful control over the system. For the accessibility library, we focused on whether an application needs special privileges to use the library; if so, we focused on how such privileges

[3] For Windows, Ubuntu, and Android, we tested the latest release version as of November 2013. Attacks still work for the current release versions. For iOS, we tested iOS 6.1.4, the latest iOS 6 at the time the research was performed.

are granted.

- Using our input validation model (Figure 14), we examined the input handling process of the analyzed platform. When a check is missing or flawed, we try to launch attacks exploiting the missing or flawed check. Specifically, if the built-in natural language user interface lacks input validation or if the validation can be bypassed, we try to escalate our malware's privilege through synthetic voice. If the OS-level check is missing and there is a security mechanism that requires user consent, we try to escalate our malware's privilege by spoofing the mechanism with synthetic input. If the OS-level check is *not* missing, we assess whether its access control policy is consistent with other security mechanisms; if not, we evaluate what new capabilities become available. If the application level check is missing or flawed, we examine whether accessibility support provides us new capabilities.
- Using our output validation model (Figure 15), we examined the output handling process of the analyzed platform. If the OS-level check is missing, we try to read the UI structure of other applications. If the application level check is missing, we examine whether new capabilities become available. In particular, since most of the displayed information is available through screenshots, we try to steal a password because it is usually not displayed in plaintext. We assume obtaining any other (potentially sensitive) information as plaintext via AT is no harder than reading a password.

4.4.2 Availability of Accessibility Features

Table 5 summarizes the availability of a natural language user interface and accessibility libraries on the four platforms. Natural language user interfaces are available on all platforms except Ubuntu; accessibility libraries are available on all studied platforms [4].

For natural language user interfaces, both Speech Recognition and Touchless Control

[4] On iOS, there is no accessibility library, but the UIAutomation framework provides most capabilities that we require.

Table 5: A list of available accessibility libraries and natural language user interfaces on each platform. * indicates the feature requires special setup/privilege.

Platform	Natural Language User Interface	Accessibility Libraries
Windows	Speech Recognition*	UIAutomation
Ubuntu	None	ATK, AT-SPI
iOS	Siri	UIAutomation*
Android	Touchless Control*	AccessibilityService*

for the Moto X[5] require initialization (training) before first use. Siri can be enabled without any setup. Although Speech Recognition on Windows requires initialization, this step can be bypassed by modifying the values of a registry sub-key at `HKEY_CURRENT_USER /Software/Microsoft/Speech`. Since this key is under `HKEY_CURRENT_USER`, it is writable by any unprivileged process.

For accessibility libraries, both desktop environments (Windows and Ubuntu Linux) have no privilege requirements for using the libraries. Thus they are available to any application.

On iOS, the UIAutomation framework, though not a full-fledged accessibility library, provides the functionality to send synthesized touch and button events. Since this framework is part of the private API set, its usage is forbidden by apps in the Apple App Store. However, as demonstrated in an attack to iOS [157], the enforcement can be bypassed.

Unlike other platforms, Android's accessibility library (AccessibilityService) is available only after the following requirements are met: first, the app must declare the use of the permission `BIND_ACCESSIBILITY_SERVICE`. Second, the user must explicitly enable the app as an accessibility service. When changing accessibility settings, a user is prompted with a dialog that displays what kind of capabilities will be granted to the AccessibilityService, which is very similar to the app permission system. Nonetheless, users are prone to enable permissions when apps provide step-by-step instructions. In particular, we find that there are more than 50 apps on the Google Play store that declare use of permissions for AccessibilityService, and two of them [63, 132] have been downloaded by more than ten

[5] For the natural language user interface on Android, we try to analyze Touchless Control which is only available on the Moto X, due to the lack of a privileged natural language user interface in Android by default.

Table 6: The status of input validation on each platform. * indicates the check enforces a security policy that is different from other security mechanisms.

Platform	Assistive Tech. Check	OS Level Check	Application Level Check
Windows	None	UIPI*	None
Ubuntu	N/A	None	None
iOS 6	None	None	None
Android	Authentication	Permission*	None

million users combined.

4.4.3 Vulnerabilities in Input Validation

Table 6 summarizes the examination results of each platform when checked against our input reference model (Figure 14). There are two common problems across all analyzed platforms.

Missing or flawed input validation within AT. Natural language user interfaces usually have more privileges than normal applications; most of them lack authentication for voice input. Moreover, some accept self-played input (sending audio from the built-in speaker to a microphone), making it possible to inject audio input through text-to-speech (TTS). Although Touchless Control on the Moto X tries to authenticate its input, the authentication can easily be bypassed with a replay attack. As a result, an attacker can obtain the privileges of the natural language user interface (attack #1, #5, #9).

Control of other applications. At the application level, no platform provides a precise way to check whether the input event is from the hardware or from the accessibility library. Moreover, at the OS level, although Windows and Android have access controls for AT, their protections are not complete. This allows a malicious AT to control most applications the same way as a human user would. Specifically, a malicious AT can send input events to make other applications perform security sensitive actions (attack #4, #6, #7, #10) and spoof security mechanisms that require user consent (attack #2, #3, #8).

Implementation of attacks. We tested all Windows-based attacks by implementing proof-of-concept malware. For controlling apps on Ubuntu Linux, iOS 6, and Android, we checked the capability of sending synthetic input to other applications by writing sample code for sending basic user interactions such as clicking a button, and writing content into a textbox. For iOS, we also wrote code to test for special UI windows such as passcode lock, password dialog, and remote view. For Touchless Control, we implemented sample malware that records sound in the background; we then sliced the authentication phrase from it manually and replayed the slice within the malware. For Siri, we manually performed the same attack.

4.4.3.1 Windows

The OS-level check applied to the accessibility library on Windows is called User Interface Privilege Isolation (UIPI) [116]. UIPI is a mandatory access control (or mandatory integrity control (MIC) in Microsoft's terminology) that sets an integrity level (IL) for every process and file, and enforces a relaxed Biba model [21, 115]: no write/send to a higher integrity level. The integrity levels (IL) are divided into five categories: *Untrusted*, *Low*, *Medium*, *High*, and *System*.

Regular applications run at Medium IL, while processes executed by an active administrator runs at High IL. As an MIC, the IL of a process is inherited by all of its child processes and takes the minimum privilege when two or more ILs are applied on the process.

UIPI prevents attackers from sending input to higher IL processes. For example, malware cannot spoof UAC through a synthesized click because normal programs including malware run at either Medium IL (when launched by the user) or Low IL (when launched by a browser, i.e., drive-by attacks), while the UAC window runs at System IL. Furthermore, malware cannot take control of applications that are executed by the administrator, which has a higher IL (High IL).

Unfortunately, the protection provided by UIPI is not complete: since most applications

are running at the same Medium IL as malware, UIPI allows malware to control most other applications via AT. Furthermore, the lack of security checks at the assistive technology and application levels results in more vulnerabilities: missing input validation in the built-in natural language user interface allows privilege escalation attacks through Speech Recognition (attack #1); missing application level checks enables escalation of privilege (attack #2), and theft of user passwords (attack #3).

Attack #1: privilege escalation through Speech Recognition. Control of Speech Recognition is security-sensitive for several reasons. First, although there is a setup phase, it can be bypassed as mentioned in §4.4.2. After setup, any process can start Speech Recognition. Second, Speech Recognition always runs with administrative privilege (High IL) regardless of which process runs it. This allows it to control almost all other applications on the system, including applications running with administrative privileges. Because of these "features" of Speech Recognition and the problems mentioned previously (i.e., no input validation, and accepting self-played voice), malware running at Medium or even Low IL can escalate itself to administrative privilege through synthetic voice.

Figure 16 shows the workflow of the privilege escalation attack from a Medium IL malware. The first step is to launch Speech Recognition through `CreateProcess()` with the argument `sapisvr.exe -SpeechUX`. Second, the malware launches the `msconfig.exe` application through `CreateProcess()`. Since `msconfig.exe` is an application for an administrator to manage the system configuration, it automatically runs at High IL. While malware cannot send input events to this process (prevented by UIPI), Speech Recognition can. After launching `msconfig.exe`, the malware can use voice commands to launch a command shell by choosing an item under the Tools tab of `msconfig.exe`. This is accomplished by playing a piece of synthetic speech "Tools, Page Down, Command Prompt, Launch!". Once the command shell that inherits the High IL from `msconfig.exe` is launched, the malware then says "cd" to its directory, says its own executable name and "Press Enter" to be executed

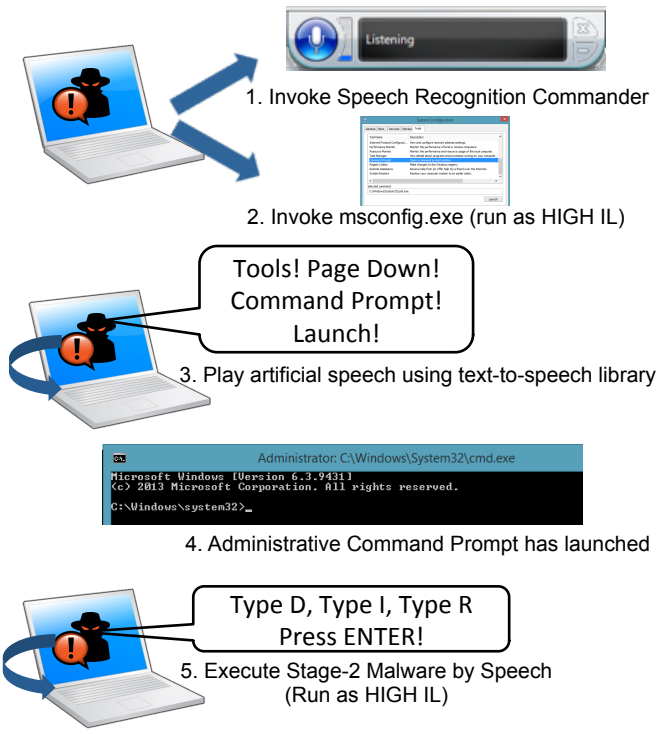

Figure 16: The workflow of privilege escalation attack with Windows Speech Recognition.

with administrative privileges.

Attack #2: privilege escalation with Explorer.exe. `Explorer.exe` is a special process in Windows that has higher privilege than its running IL. Unlike other Medium IL processes, `Explorer.exe` has the capability of writing to High IL objects such as the `System32` directory. Although this capability is protected by a UAC-like dialog (Figure 17), i.e., `Explorer.exe` asks for the user confirmation before writing to a system directory of Windows, the dialog belongs to `Explorer.exe` itself. Since this action requires user consent, the application should check whether the input comes from the user or AT. However, there is no such check. As a result, malware can overwrite files in system directories by clicking the confirmation dialog through the accessibility library.

Some system applications in system directories are automatically escalated to the administrative privilege at launch. On Windows, when a process tries to load a DLL, the dynamic linker first looks for the DLL from the local directory where the executable resides.

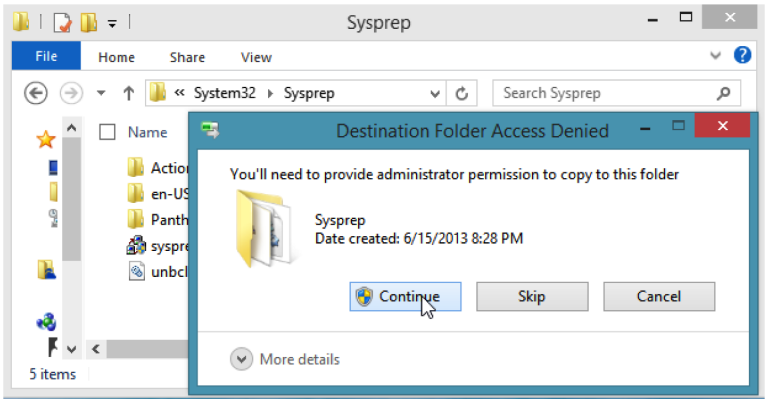

Figure 17: A dialog that pops-up when `Explorer.exe` tries to copy a file to a system directory. The dialog runs at the same Medium IL as `Explorer.exe`. Thus, any application with Medium IL can send a synthetic click to the "Continue" button and proceed with writing the file.

Once malware injects malicious DLLs into the directory containing these applications, it can obtain the administrative privilege when the applications are run, thus bypassing UAC. An example of such an application is `sysprep`, which will load `Cryptbase.DLL` from the local directory. By sending synthetic clicks to `Explorer.exe` and injecting a malicious `Cryptbase.DLL`, malware can achieve privilege escalation.

Attack #3: stealing passwords using Password Eye and a screenshot. On Windows, passwords are protected in several ways. They are not shown on the screen; and even with real user interactions, the content in a password box cannot be copied to the clipboard. Furthermore, as will be described in detail in Table 7, it is also not possible to retrieve password content directly through the accessibility library. However, the lack of input validation on the password box UI component opens up a method of stealing the plaintext of a password.

Starting with Windows 8, Microsoft introduced Password Eye as a new UI feature to give visual feedback to users to correct a typo in a password input box (Figure 18). This "Eye" appears when a user provides input to a password box, and clicking it will reveal the plaintext of the password. Unfortunately, since Password Eye cannot distinguish hardware input from a synthetic input, malware can click it as long as UIPI permits. Again, since

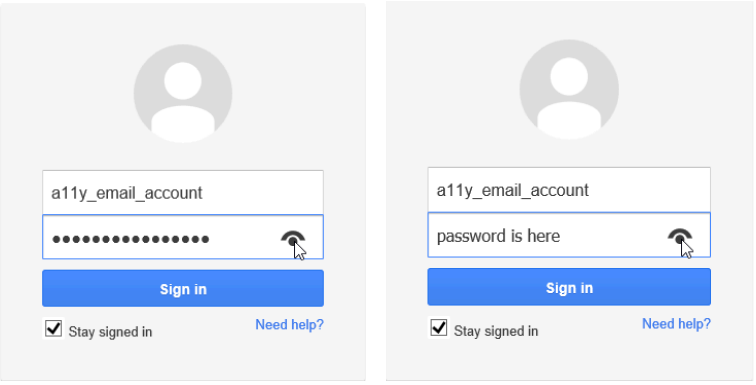

Figure 18: Password Eye on the Gmail web application, accessed with Internet Explorer 10. In Windows 8 and 8.1, this Eye is attached to password fields not only for web applications but also for regular applications. By left-clicking the Eye, the box reveals its plaintext content.

most applications run at the same IL as malware, malware can send a left-click event to reveal the content of the password dialog (Figure 18) and can extract it from a screenshot.

4.4.3.2 Ubuntu Linux Desktop

Since Ubuntu does not have a built-in natural language user interface, we only consider the attacks enabled by missing checks in the OS or an application. The missing check at the OS level allows malware to control any application and thus break the boundary enforced by other security mechanisms (attack #4). The missing check at the application level does not provide additional capabilities beyond those already provided by the missing OS level check.

Attack #4: bypassing the security boundaries of Ubuntu. Since neither the OS nor applications authenticate input, malware can send a synthetic input to any application in the GUI, i.e. the current X Window display. The display here does not mean the physical display (i.e., a monitor screen) of the device; rather, it refers to the logical display space (e.g., `:0.0`) of the X Window Server.

In this setting, the lack of security checks for input breaks two security boundaries in Ubuntu. The first violation is regarding user ID (UID) boundaries. Regardless of the UID

of the display service, a launched process will run with the UID of the user who launched it. For example, if a non-root user runs a GUI application with `sudo` (e.g., `sudo gparted` or a GUI shell with root privileges), the application runs in the same display space of the non-root user account, even though it runs as the root UID. Since AT-SPI allows control of any application running on the logical display space, malware with a non-root UID can send synthetic input to control other applications, even those with root privileges.

Second, process boundaries can be bypassed by sending a synthetic input. Starting with Ubuntu 10.10, Ubuntu adopted the *Yama* security module [36] to enhance security at the kernel level. In particular, one feature in *Yama* prohibits a process from attaching to another process using the `ptrace()` system call, unless the target process is a descendant of the caller. Thus, a process cannot attach or read/write other processes' memory if the target process is not created by itself or its descendants. However, malware can bypass this restriction: it can write values or perform UI actions to change application status through synthetic inputs or interfaces available by AT-SPI such as `settextvalue()` and `invokemenu()`.

4.4.3.3 iOS

iOS 6 lacks security checks at all levels. Missing input validation in its natural language user interface, Siri, allows an attacker to abuse its privileges to perform sensitive operations and access sensitive information (attack #5). Furthermore, missing OS-level checks allows malware to 1) bypass sandbox restrictions to control other apps (attack #6), 2) spoof the remote view mechanism to programmatically authorize access permissions to sensitive resources (attack #7), and 3) bypass password protection (attack #8).

Finally, since there are no available checks at the application level, synthetic input from a malicious app cannot be prevented or detected by the targeted application.

Attack #5: bypassing passcode lock using Siri. [6] iOS allows several security-sensitive

[6] We note that this attack on Siri was not originally discovered by us. The attack has been publicly known since September 2013 [46], but we include this in this chapter due to the importance of its security implications

actions to be carried out through Siri even when the device is locked with a passcode. Such actions include making phone calls, sending SMS messages, sending emails, posting messages on Twitter and Facebook, checking and modifying the calendar, and checking contacts by name. Since there is no input validation, any attacker who has physical access to the iOS device can launch the attack without any knowledge of the passcode.

Attack #6: bypassing the iOS sandbox. App sandboxing [10] in iOS enforces a strict security policy that strongly isolates an app from others. The data and execution state of an app is protected so that other apps cannot read or write its memory, or control its execution (e.g., launching the app). However, the lack of OS-level security checks on accessibility makes it possible for malware to control other apps by sending synthesized input.

With synthetic touch, malware can perform any actions available to a user, such as launching other apps, typing keystrokes, etc. That is, malware can *steal* capabilities of other apps across the app sandbox.

Attack #7: privilege escalation with remote view. . In addition to app sandboxing, iOS protects its security sensitive operations with the remote view mechanism [18]. Protected operations include sending email, posting on Twitter or Facebook, and sending SMS. Remote view works as follows: when an app tries to access any protected operation, the underlying service (which is a different process) pops up a UI window to seek user consent. For example, if an app wants to send an email, it invokes a remote function call to the email service, which would then pop up a confirmation window. The email message can only be sent after the user clicks the "Send" button in the pop-up window.

Remote view is considered an effective defense mechanism to prevent misuse of sensitive operations. However, the lack of input validation in iOS allows malware to send synthetic touches to spoof user input to remote view and execute these privileged operations.

---

on built-in AT.

Figure 19: Screenshot of passcode and password input in iOS. For passcode (left), typed numbers can be identified by *color differences* on the keypad. For the password (right), iOS always shows the last character to give visual feedback to the user.

Attack #8: bypassing password protection on iOS.

Another protection mechanism in iOS is passwords. This is utilized in two system apps: the lock screen and the App Store.

The lock screen prevents any unauthorized access to the device and is applied not only to UI events, but also to security data such as KeyChain and encrypted files. Moreover, once the screen is locked, all touch events are blocked; thus malware is no longer able to manipulate apps other than the lock screen.

The App Store asks for an Apple ID and password for each purchase. Although malware can generate "clicks" to initiate the purchase, without knowing the password, it is not possible to finish the transaction.

Unfortunately, since iOS always displays the last character of a passcode/password in plaintext (Figure 19) and background screenshots can be taken through the private API call `createScreenIOSurface` in `UIWindow` class, it is possible to steal the user's passcode and password. With a stolen password, since both the lock screen and the App Store accept synthesized input, malware can programmatically unlock the device and make malicious transactions.

4.4.3.4 Android

The Android platform has the most complete input validation among the four evaluated platforms. First, *Touchless Control* [123], a natural language user interface for the Moto X, utilizes voice authentication: the user is required to register his/her voice with Touchless Control at first boot; the app then constantly monitors microphone input for the fixed authentication phrase "OK Google Now" from the user. Once the command is captured, it checks whether the phrase matches the voice signature extracted from the registered phrase; if so, it then launches the Google Now application to execute a voice command. Nonetheless, like other non-challenge-response-based authentication, this voice authentication is vulnerable to *replay attacks* (#9).

Second, as discussed in §4.4.2, Android requires explicit user consent to acquire accessibility capabilities. However, its protection is incomplete. Specifically, Android has no runtime security check for AT. Once an app is allowed to be an AT, it can leverage the accessibility library to create a new inter-process communication (IPC) channel that is not protected by the ordinary Android permission system (#10). As a result, a malicious AT can easily achieve the same effect as capability leakage attacks [56, 31, 41, 70, 164] and information leakage attacks [84, 167]. Moreover, unlike UIPI, Android's OS level access control on accessibility does not protect system apps. In particular, we found that our malware can change system settings through AT, which offers us many powerful capabilities.

The only missing check in Android is at the application level. Similar to the iOS case, we did not find new capabilities beyond what is enabled due to inconsistent OS-level checks.

Attack #9: bypass Touchless Control's voice authentication. Fragile authentication for AT leads to a vulnerability in Touchless Control on the Moto X. In particular, voice authentication can be bypassed by a replay attack shown in Figure 20. First, an attacker can build malware as a background service that constantly monitors sound from the microphone. As the phrase "OK Google Now" is the only authentication phrase, the user is likely to repeat it frequently. The malware can easily record the authentication phrase. Once

Figure 20: The workflow of the attack on the Moto X's Touchless Control. Malware in the background can record a user's voice, and replay it to bypass voice authentication.

recorded, the malware can play the recorded phrase through the device speaker to activate Touchless Control. Since Touchless Control accepts self-played sound from the speaker to the microphone, it subsequently launches Google Now. After this, the malware can play arbitrary commands using the default TTS library for Google Now. Since there is no further authentication for the command phrase, the malware can utilize a variety of commands to make phone calls, send SMS, send email, set alarms, and launch applications.

Attack #10: bypassing Android sandboxing and more. Sandboxing in Android [6] provides isolation between apps to protect memory and filesystem access, and prohibits an app from interfering with the execution of other apps. Furthermore, its permission system restricts an app's access to sensitive resources.

However, once an app is activated as an AT, there are no further restrictions. A malicious AT can then read UI structure (including location, type, text, etc.) of the whole system and deliver user actions to any UI element, such as the click of a button, a scroll up or down, a

Table 7: The status of output validation on each platform. * means the check enforces an inconsistent security policy.

Platform	Reading of UI Structure	Password Protection
Windows	UIPI	Yes
Ubuntu	None	Yes*
iOS	N/A	N/A
Android	None	Settings*

cut/copy/paste of text, a change of focus, and expand/dismiss of UI. Therefore, malware can control other apps as if it is the user. Malware can abuse the permissions of other apps, e.g., even without network permission, our malware can control the Gmail application to exfiltrate stolen data.

Additionally, malware can change system settings such as user-configurable settings, and install/uninstall apps. Moreover, malware can programmatically enable developer mode (e.g., by sending seven synthetic clicks) which can put a device at risk for further infection.

4.4.4 Vulnerabilities in Output Validation

Table 7 summarizes the evaluation results of each platform compared against our output reference model in Figure 15. iOS does not support alternative output, so its result is omitted in this section.

Across all platforms, only Windows enforces an OS-level check (UIPI) for output. However, since UIPI does not have any protection among applications in the same IL, Windows suffers from the same UI-level attacks described below.

Reading UI state of other applications. All platforms except iOS allow an AT to access UI structures. The library provides not only the metadata for the UI such as the type of element, location, and size but also the content of the UI element. Hence, a malicious AT can monitor other applications in a fine-grained manner. For example, malware can detect the current state of the target application using 1) available UI structures, 2) UI events such as change of focus, movement of a window, and change of contents, and 3) user interaction events. With these capabilities, malware can spy on every action a user takes, as well as

maintain an accurate status of an application.

All three platforms (Windows, Ubuntu, and Android) protect the plaintext content of the password in a password dialog box by default. However, in Ubuntu, AT-SPI fails to block all paths for retrieving the plaintext of the password (#11). Android can be configured to allow reading keystrokes on password dialog boxes; this can be enabled by malware implemented as an AT (as mentioned in #10).

Implementation of attacks. For extracting passwords in Ubuntu (attack #11), we implemented proof-of-concept malware that looks for authentication windows, obtains the plaintext, and prints out the plaintext on the console using AT-SPI. For attack #12, we implemented malware that enables the speaking of passwords via accessibility services and registers itself as the TTS subsystem for the accessibility service. In this two-fold manner, malware can receive and transmit the contents of a password to an attacker.

4.4.4.1 Windows

With UIPI, Windows is the only platform where the OS applies access control to the reading of UI structures. Although UIPI prohibits accessing the structures of an application that has higher IL than the caller, access on the same or lower IL is still permitted.

The application level output check exists for password boxes by default, which disallows 1) obtaining the password via `WM_GETTEXT` or `ValuePattern` in UI Automation, and 2) copying the password via `WM_COPY` or by generating a Ctrl-C input event. Therefore, malware cannot steal passwords through the accessibility library.

4.4.4.2 Ubuntu Linux Desktop

In Ubuntu, the application level check for passwords exists, but its implementation (in ATK) is inconsistent with the UI (in GTK).

Attack #11: stealing sudoer passwords from authentication dialogs. On Ubuntu, we found a password stealing vulnerability using AT-SPI. The security checks at the OS level are incomplete. For a password box, there exists an API call, `gettextvalue()`, on the

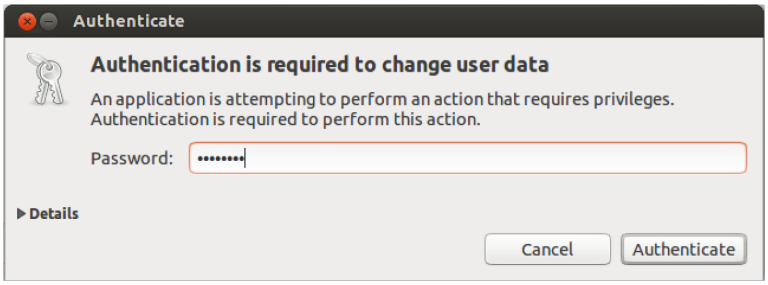

Figure 21: The administrator authentication dialog of GNOME on Ubuntu 13.10. This dialog asks for the password of the current user to gain root permissions.

Linux Desktop Testing Project (LDTP, a wrapper over AT-SPI and ATK). It throws a "Not Implemented" exception when called, meaning that reading passwords through this API is unavailable. However, AT-SPI missed security checks on a critical accessibility function of a password box: `copytext()`. Although physically or synthetically pressing Ctrl-C does not copy the value of a password box, `copytext()` from AT-SPI does copy the plaintext of a password to the clipboard. The clipboard then can provide the plaintext content of a password. Figure 21 shows a sudo dialog that is vulnerable to this attack. Once the sudoer's password is acquired in this manner, malware can easily gain root privileges.

4.4.4.3 Android

While Android prohibits reading of password content from its accessibility service, this can be disabled via user preferences. In conjunction with the vulnerability of input validation (attack #10), this restriction can easily be bypassed.

Attack #12: keylogger on Android. Although Android provides protections for accessing the plaintext of a password, incomplete protections at the OS-level lead to a vulnerability. Once an app is enabled as an AT (see Attack #10 for detail), the app can change any settings on the device without user consent. Android provides an option called "Speak passwords" in its accessibility settings. If enabled, keystrokes on a password box are delivered through the text-to-speech (TTS) processor. We register malware as a TTS output application. Once registered, the malware can receive password contents via the OS-level accessibility service.

4.5 Discussions

In this section, we explain how accessibility libraries are making it possible to implement our attacks, discuss the limitations of our attacks, analyze the root causes of the vulnerabilities, and consider open problems for future work.

4.5.1 Complexity of Accessibility Attacks

As we mentioned in section §4.2, accessibility libraries provide three capabilities: 1) obtaining events representing UI change, 2) providing a way of programmatically probing/accessing UI widgets, and 3) synthesizing inputs to UI widgets.

With these functionalities, an attacker can create malware capable of performing successful attacks with a degree of relative ease when compared to other non-AT methods that achieve the same ends. As an example, we will describe how the "Password Eye" attack (#3) can be implemented using accessibility libraries. To achieve the "Password Eye" attack, malware needs to: 1) detect when the user types a password, 2) identify the UI "eye" and click on it, and 3) locate the password field to grab its text in a screenshot. To determine whether the user is typing a password, we can use the first capability of the accessibility libraries to keep track of which UI component is currently focused. In particular, on the Windows platform, this can be easily achieved by registering an event handler in the platform's accessibility framework that receives the focused UI element at any change of focus. After being handed a focused element, we can check whether the element is a password box with an "eye" by assessing its properties reported by the accessibility libraries. For example, a `TRUE` value of `isPassword` property indicates a password box. Once we determine that the focused element is a password box, we can use the second capability of the accessibility libraries to get the "eye" button. In particular, since we know the relative position of the focused text box and the "eye" button, we can walk the UI widget tree provided by the accessibility library and calculate the position of the "eye". Then, we can use the third capability to click it. Finally, the handle to the focused password box we obtained in the

first step can also be used to retrieve the location of the box on the screen and also allow us to grab the actual password typed from a screenshot. A point worth noting here is that developing attacks using accessibility libraries is very similar to how one manipulates DOM (Document Object Model) objects using Javascript in a web page.

One may point out that the same attack can be achieved on the Windows platform by sending traditional Windows Messages (such as `WM_CLICK`), or using tools such as AutoIt. However, we argue that the use of the accessibility library greater ease and reliability. In particular, without the first capability of the accessibility libraries, one may need to constantly probe the current state of UIs to determine if the user is typing a password. Secondly, while it might be trivial to use a hardcoded coordinate to click the "eye" button in a testing environment such as AutoIt, this strategy will be very fragile in a real attack; factors such as variation in screen size and resizing/moving of the target window may break the hardcoded approach in a real attack. Using hardcoded locations to extract a password from screenshots will face a similar issue. Even though it may be possible to reliably implement our attacks without accessibility libraries, this implementation would be more complex and require greater effort on the part of the author.

4.5.2 Limitations of the Attacks

Since attacks through accessibility libraries perform actions over user interfaces, they have an inherent limitation in that they are not stealthy. For example, if the target application is running in the foreground when an attack is unfolding, the user may recognize visual cues of the attack, such as button presses, an opening of a new UI window, etc. Furthermore, attacks via voice commands play sounds, and are thus audible; or they fail if the speaker is turned off.

However, we argue that these attacks *can* be launched in a stealthy way. First, malware can detect whether the user is using the device or not. For desktop machines, the presence of a user can be detected by monitoring physical keystroke or mouse movement. Malware can

exploit a time period when the user is absent to launch UI-intensive attacks. If necessary, the malware can blank the screen when launching the attack, because screen sleep after some period of non-use is a natural and expected behavior of the system. For mobile devices, prior research works [135, 165, 73] discussed how to track the user's behavior using an app on the device. With the help of various sensors, such as the camera, face proximity detector, GPS location, accelerometer, etc., malware can determine when the user is not watching the screen, away from the device, or when the device is in the user's pocket. It can then launch an attack without being exposed.

Second, UI actions can be delivered in the background for some platforms. Thus, an attack can be carried out even when the user is actively using the device. In Windows, once a handle to a UI widget is obtained while it is in the foreground, it can still be manipulated even when it is in the background or minimized. In Linux, probing the UI of a minimized application is possible. Furthermore, in the worst case, malware can move a window to nearly off the screen, so that the user does not notice any UI change. In our experiment, if any pixel of an app is visible on the screen [7], there is no limitation on probing or performing actions on it.

Third, it is possible to make the attacks on natural language user interfaces stealthy with the help of hardware. Common audio devices such as the Realtek HD Audio device and other sound card devices' drivers provide functionality called `Stereo Mix`. `Stereo Mix` sends the output of system sound to an internal microphone input. Enabling this functionality does not require any special privilege. Malware can play audio *internally* to deliver text-to-speech audio to a natural language user interface. The attack succeeds without outputting audio to speakers and also works when there is no speaker device at all.

Finally, our experience with OS vendors shows that these threats will be taken seriously. In the May of 2013, before presenting an attack [103] that takes advantage of private APIs for synthesizing touches and taking screenshots on iPhone, we informed Apple of our

[7] For example, while all other pixels are invisible, only one pixel of the window is visible.

```
// On real touch event
public boolean onTouchEvent(MotionEvent event) {
  switch (event.getAction()) {
    case MotionEvent.ACTION_UP:
    {
      // ...
      // performClick() is called to handle real click event
      performClick();
      // ...
    }
  }
}

// On a11y request for click
boolean performAccessibilityActionInternal(int action,
                                    Bundle arguments) {
  // ...
  switch (action) {
    case AccessibilityNodeInfo.ACTION_CLICK:
    {
      if (isClickable()) {
        // the same performClick() is invoked to handle a11y request
        performClick();
        return true;
      }
    } break;
  }
  // ...
}
```

Figure 22: Code that handles the real input (*above*), and code that handles the a11y input (*below*) for click, in `View.java` of Android. The same function `performClick()` is used to handle both requests.

attack. In the August of 2013, the exploited vulnerabilities were removed from the then newly-released iOS 7.

4.5.3 Root Causes, and Design Trade-offs

We strongly believe that to eliminate a11y related vulnerabilities fundamentally, a new architecture for providing accessibility features is necessary. However, proposing such an architecture is out of the scope of this work; instead, we present the findings of our root cause analysis to illustrate why security checks spread across the AT, OS, and application tend to fail, and to show some of the trade-offs taken in the current implementation of accessibility features.

The first identified root cause is the emphasis on availability/compatibility of a11y support in all the studied systems. In every case we have studied, native UI widgets include logic to handle requests from accessibility libraries, and UI widgets provided by OS are

```
1  static void gtk_entry_copy_clipboard (GtkEntry *entry) {
2    GtkEntryPrivate *priv = entry->priv;
3    // ...
4    // ### security check for password box ###
5    if (!priv->visible)
6    {
7      // do not copy text to clipboard
8      gtk_widget_error_bell (GTK_WIDGET (entry));
9      return;
10   }
11   // ...
12 }
```

Figure 23: Code that handles copying of text (pressing Ctrl-C) in GTK. Inside the function, GTK checks the security flag `priv->visible` to decide whether or not to provide selected text to the clipboard. If `GtkEntry` is set as a password box (if the flag is true), then the text will not be copied.

usually built to reuse the same interfaces/channels to handle both real user inputs and a11y inputs. As a result, it is very hard for an application to distinguish a11y inputs from real user inputs. This design choice enables many attacks by accepting and processing synthesized input as if it is a real input (A2[8]). For instance, in Android, physically tapping a UI widget with a finger will invoke the `performClick()` function. Equally, on an a11y request, the same `performClick()` function is invoked (see Figure 22 for details). In Windows, just like the real user input, clicks generated by `UIAutomation` are delivered as a Windows Message `WM_CLICK`. Similarly, for Ubuntu and iOS, a11y requests take the same path as I/O requests within the UI widget. While this means all applications that use the native UI widgets automatically and naturally work with the requests from accessibility libraries, such design also imposes a default security policy that makes every widget available to all ATs. As we can see in attack #2 and #3, this is too permissive policy. Furthermore, in all the studied systems, if the application/UI developers were to instead implement their own policy regarding how an application should process requests from accessibility libraries, they would have to implement their own UI widgets (usually by "subclassing" the native ones), and this comes with a non-trivial cost.

Second, from both technical and economic perspectives, it is challenging to perform complete validation and authentication for certain inputs introduced by AT. As a result, new

[8]Please refer to §4.3.1 New Attack Paths for details.

```
// A11y code snippet
void atk_editable_text_copy_text (Editable e, int start, int end) {
  AtkEditableText *text;
  // ...
  *(iface->copy_text) (text, start_pos, );
  // calls gtk_entry_accessible_copy_text()
}

static void gtk_entry_accessible_copy_text(AtkEditableText *t,
                                           int start, int end) {
  GtkEditable *e;
  // ...
  gchar *str = gtk_editable_get_chars (e, start, end);
  // ...
}
// A11y code end, calls functions in Gtk UI

// Gtk code snippet
gchar* gtk_editable_get_chars (GtkEditable *e,
                               int start, int end) {
  return (editable)->get_chars (e, start, end);
  // calls gtk_entry_get_chars()
}

// Final function that returns text content
gchar* gtk_entry_get_chars (GtkEntry *e, int start, int end) {
  gchar *text;
  text = gtk_entry_buffer_get_text (get_buffer (entry));
  // ### no security checks at all on getting text ###
  return g_strndup (text + start_index, end_index - start_index);
  // return text without checking priv->visible
}
```

Figure 24: Code that handles an accessibility request (ATK) for copying text. ATK internally calls a function of a module in GTK that supports accessibility. The module then calls a function that directly interacts with the UI widget (GTK functions). However, the module `GtkEntryAccessible` calls a different function `gtk_editable_get_chars()`, which misses required security checks of the password box.

attack vectors become available due to missing security checks on processing input (A2) and output (A3) requests from ATs or accessibility libraries. For example, in attack #11, simply pressing Ctrl-C will call `gtk_entry_copy_clipboard` in which there is a security check for preventing the text in a password field from being copied (see Figure 23 for details). However, a different function `copytext()` will be executed in ATK, which takes a different execution path without security checks, potentially leading to password leakages. We suspect that the ATK code was added to the OS by a group of developers who were not aware of the principles of input validation and complete mediation, or that the ATK code was added to the OSs only recently and has thus not been through rigorous security code review and testing when compared to older portions of the OS.

There are also technical and economic reasons for a lack of validation and authentication. For example, for the cases of attack #1 and #9, the AT needs to check whether the voice input actually comes from a real user, and also needs to further authenticate the authorized user. Voice based validation and authentication require non-trivial technical support, with potentially high research and development costs.

Finally, to improve the usability of ATs, OSs usually have weak access control on accessibility libraries; while this makes the installation and use of ATs (their intended purpose) easy, it is not a good security practice. In particular, accessibility libraries can usually be accessed by any application on a system. For example, in Windows, iOS 6, and Linux, any program can be an AT without any authorization. This also opens paths for attacks so that any (malicious) program can abuse accessibility functionalities to launch the attacks described in this chapter. The exception is Android; it has a setup menu for enabling an app's use of the accessibility library, though this check is only performed at initial app setup.

4.5.4 Recommendations and Open Problems

Based on the root cause analysis in §4.5.3, we present recommendations on how to alleviate (if not eliminate) the security risks created by the a11y support. Our recommendations are intended to work with the current architecture for supporting accessibility features, and thus are limited by the inherent difficulties that come with this architecture; nonetheless, we believe they will help the community to improve security for a11y before the introduction of a complete a11y security policy occurs. We will also discuss some open problems involved in implementing these recommendations.

Our first recommendation is to have fine-grained access control over which program can access specific functionality of the accessibility library. From our study, we find that both Linux and iOS have no such access control at all, while Windows allows all programs to use the accessibility library to control/read the content of any other program with the same integrity level. Android appears to be the only system that has access control policy specific for the accessibility library: the user has to explicitly grant the AT the privilege to use the accessibility library. However, once this privilege is granted, the AT has full access to all the capabilities of the accessibility library. In many cases, this violates the principle of least-privileged access. For example, a screenreader will only need to read the content of other apps through the accessibility library, but it does not need to be able to interact with other apps. Based on this observation, we recommend the privilege of using the accessibility library be at least split into two, one for reading the content of other apps and one for the more privileged capability of interacting and controlling other apps. While this may present an extra hurdle for users who need AT, it will only incur a one-time setup cost, which we feel is an acceptable trade-off for the extra security against misuse of the accessibility library.

Our second recommendation is to provide mechanisms for a UI developer to flag how different widgets in their UI will handle various requests from the AT, rather than requiring the UI developer to handle this task themselves. For example, in many UI libraries, a developer can flag a text field as a password field, and the underlying logic of the UI will

make the content in the field invisible to both the display and the ATs. However, this generally appears to be the only instance of such a flag, and it only applies to text fields. We believe more such flags should be available to specify various a11y related security policies, and such flags should be made applicable to various kinds of widgets (e.g. attack #2 and #3 can easily be eliminated if a security flag is applicable to buttons). As future work, we will study what kind of a11y related security policies UI developers usually need to specify, and what language features are needed for specifying such policy as attributes of widgets in the UI.

Our final recommendation can be considered a new security component in the current a11y architecture, and can significantly limit the damage caused by exploitation of a11y-related vulnerabilities. We propose to extend accessibility support to user-driven access control mechanisms like UAC in Windows or Remote View in iOS. While this recommendation may not be directly derived from our root cause analysis, we believe it will fundamentally eliminate many a11y related security issues discussed in this chapter. In particular, OS vendors should develop versions of access control mechanisms to support various disabilities. For example, for visually impaired users, the system can read out (through the speaker) the message seeking permission, and have the user confirm or abort by clicking the "F" or "J" button on the keyboard (which are tactilely different from all other keys on the keyboard), and for the users who lack fine motor skills, the permission granting can be driven by voice recognition. We note that while this approach is not general enough to support the need for all users with different kinds of disabilities, it will significantly improve the security for all users that are covered. Furthermore, in the case of voice recognition, the introduction of a mechanism specifically designed for seeking vocal permission may significantly simplify the task of authenticating user input (only "yes" or "no" need be verified, rather than performing general voice recognition), and thus move the burden of performing voice recognition from the AT developer to the OS vendor (who may have more resources to research and develop a mechanism that is robust against attack #1 and #9).

Finally, we acknowledge that our analysis requires significant manual effort and reverse engineering work and thus is not exhaustive. We will leave it as an open problem to design systems that can automatically find a11y related vulnerabilities. We believe this will be a challenging problem for the several reasons. First, automatically detecting a11y functions and analyzing their related vulnerabilities requires whole system analysis. Since an a11y request is regarded as an I/O event, it is processed asynchronously. As a result, it is very hard to find entry points. The complicated execution of a11y logic extends to many different low-level modules, which usually make use of many (function) pointers. Proprietary OSs do not provide source code, and so researchers can only perform analysis with the compiled binary, which makes the task even harder. Second, unlike general programming errors, confirming a11y related vulnerabilities requires a deep understanding of the semantics of an application, which significantly limits the scalability of such analysis. We hope that our work can motivate further studies toward this direction.

4.6 Related Works

Attacks on Windows. In 2007, it was reported that an attacker could control a Windows Vista machine by playing an audio clip to Speech Recognition [127]. However, since the attack could not bypass UAC and assumed Speech Recognition was already enabled, it was considered a minor bug at that time. Compared to this attack, our attack (attack #1) does not require Speech Recognition to be enabled before the attack, and we can bypass UAC on Windows 7 through 8.1 (due to policy changes in UAC [122]).

Just before the release of Windows 7, there was a UAC bypass attack [42] that exploited the special capability of `Explorer.exe` to write to system directories. In this attack, a malware process will attach to `Explorer.exe`, inject code, and exploit its capability to write to system directories. Our attack #2 follows the same strategy, but instead of using low-level function `WriteProcessMemory()` to inject code into `Explorer.exe`, we used the accessibility library to simply click the "OK" button.

Attacks on iOS. Recently, it was reported [46] that Siri in iOS 7 could be exploited to bypass the lock screen and send email, SMS, post on Twitter and Facebook, make phone calls, etc. We referred to this attack as attack #5 in the vulnerability section.

Although the accessibility library is a private API that is not usable by regular app developers, the threat is real. Last year, an attack [157] showed that it is possible to circumvent the Apple App Store review process by successfully publishing an App Store app that invoked private API calls.

Attacks on Android. In Android, there have been many attacks on the permissions [56, 31, 41, 70, 164] and private information [84, 167] of an app that demonstrate data leakage through Android's IPC channel. To address these problems, researchers have proposed a number of mechanisms [49, 70, 27, 105]. Unfortunately, since all of the proposed mechanisms were focused on the official IPC channel, they are not able to prevent attacks through accessibility libraries. Furthermore, our attacks can steal the capabilities and private information of other apps.

4.7 Summary

In compliance with the amendment to the Rehabilitation Act of 1973, software vendors have been continuously adding accessibility features to their OSs. As the technology advances, accessibility features have become complex enough to comprise a complete I/O subsystem on a platform. In this chapter, we performed an analysis of the security of accessibility features on four popular platforms. We discovered vulnerabilities that led to twelve practical attacks that are enabled via accessibility features. Further analysis shows that the root cause of the problem is due to the design and implementation of a11y support requiring trade-offs between compatibility, usability, and security. We conclude with proposing several recommendations to either make the implementation of all necessary security checks easier, or to alleviate the impact of incomplete checks.

CHAPTER V

SGX-USB: ESTABLISHING SECURE USB I/O PATH IN INTEL SGX

5.1 Motivation

Today's system is very complex. Even a simple desktop computer consists of a huge software stack including operating system, device drivers, system daemons and other applications, etc. Thus protecting the entire software stack of a system is extremely difficult. One promising approach to protecting a system is to reduce the attack surface by isolating the execution runtime into a separate environment. History of building secure OS and hypervisors have evolved into many software-based approaches [113, 137, 112, 97, 125, 99, 30] to provide the trusted execution environment (TEE) in commodity systems. Recently, Intel has introduced a new hardware extension, Intel Software Guard Extension (SGX) [75], which provides a hardware-based TEE as an enclave. While software-based TEEs still require either a trusted hypervisor or a trusted operating system, this hardware TEE implementation offers a strong security guarantee of not trusting privileged software including operating systems and hypervisors, by isolating memory and registers at the hardware level [37, 134].

Although Intel SGX is now available in the most of the newly manufactured commodity x86 processors, this hardware TEE is still limited to server or daemon applications because Intel SGX cannot support trusted user I/O to its enclave that is running in ring 3, due to the requirement that I/O handling must be done in ring 0. In order to get benefits from Intel SGX, we design SGX-USB, which can establishe a secure I/O path between a USB device and an enclave. In particular, SGX-USB opens a secure channel that can support USB protocol, which enables supporting for variety of user I/O devices including a keyboard, mouse, camera, speaker, and display, and even for non-user-facing devices such as a disk.

To enable a secure channel, SGX-USB places a proxy device that sits in the middle of the channel and establishes a secure communication channel between an I/O device and the enclave.

These two devices establish a secure channel that can guarantee the authenticity of end points, and both the confidentiality and integrity of data channel. establishing a secure communication channel starts with the remote attestation process that authenticates an enclave and the proxy device and shares a secret between these two at the same time. After authenticating and sharing a secret, the proxy device opens a communication channel between a USB I/O device and an enclave and protect the data transmitted in the channel by using a derived encryption key from the shared secret. Throughout the remote attestation process and application of encryption over the channel, SGX-USB can guarantee the three key security properties: authenticity by remote attestation, and confidentiality and integrity by encryption. Thus, SGX-USB provides the assurance of user input and allows the enclave instance to handle commands and data from the user securely. While the currently discussed applications of Intel SGX only perform the network and the file I/O securely, this new design enables secure user I/O in the TEE so that Intel SGX can facilitate user-facing trusted applications, such as authentication manager that securely processes password. Moreover, we show that constructing an end-to-end trusted I/O channel from one user to another user over the Internet is possible with SGX-USB; for example, having a video chat over the Internet. SGX-USB can forward not only the user I/O devices but also general USB I/O devices through the established secure channel. Its overhead on the bandwidth is around 1%, and added latency is around 11 microseconds, all of which are negligible.

To summarize, we made the following contributions in this chapter:

- We design SGX-USB, which enables the trusted user I/O to an enclave of Intel SGX by establishing a trusted I/O channel between a USB device and an enclave. The design of SGX-USB ensures the authenticity of channel end points and the confidentiality and the integrity of the data that flows through the established secure channel.

- We extended the Intel SGX remote attestation process to enable authentication and secret sharing between a remote device and an enclave.
- We implemented a prototype of SGX-USB with commodity hardware, a small board computer and a desktop computer, to demonstrate the feasibility of the design of SGX-USB for securely delivering keyboard input to an enclave. Moreover, we present a potential interesting use case of SGX-USB for video chat, which establishes a user-to-user trusted I/O channel over the Internet.

The rest of chapter is structured as follows. §5.2 introduces background on Intel SGX and several related works for building trusted I/O. §5.3 gives a brief overview of SGX-USB and presents its threat model. §5.4 describes the design of SGX-USB in detail, and §5.5 demonstrates compelling use cases that are enabled by SGX-USB. §5.7 presents the evaluation result of our prototype implementation of SGX-USB. §5.8 discusses further considerations on current design and implementation of SGX-USB.

5.2 Background and Related Work

This section presents backgrounds on Intel SGX that is necessary to understand the problem that SGX-USB tackles. The contents of this section include the architecture of Intel SGX, how SGX utilizes I/O devices, and compare SGX with other TEEs.

5.2.1 Intel SGX

Intel Software Guard Extensions (SGX) [75, 80, 81, 134] is an extension of the x86 instruction set architecture (ISA), which provides trusted execution environment (TEE), called enclaves, as a user-level process.

SGX Threat Model. SGX only includes the processor hardware and the program runs in an enclave as its trusted computing base (TCB). To maintain the TCB without trusting an operating system, SGX provides an isolated memory space and execution runtime to an enclave [37]. SGX prohibits any access to the memory and registers that belong to an enclave from the execution domain other than the enclave at the architectural level. This

isolation is applied to all the other programs even including the operating system kernel, so the access to the runtime of an enclave is strictly prohibited. Moreover, SGX applies encryption to all of the data that belongs to an enclave to ensure confidentiality and integrity to protect an enclave from physical attacks such as the cold-boot attack and bus snooping attack.

Based on this isolation and encryption, an enclave can remain secure from the attacks that could be originated in operating systems, kernel device drivers, other processes, etc.

Remote attestation. In addition to the isolation and encryption, SGX provides a protocol for local and remote attestation to ensure the integrity of code and loading parameters of an enclave instance. An enclave can generate a report of its launching status that contains the measurement (i.e., hash) of loaded code in the enclave and its security parameters.

The local attestation provides a way of verifying an enclave from the other enclave on the same hardware. The report of a target enclave will be signed by a hardware key, and the other enclave can verify the report by using the same key.

The remote attestation lets a verifier placed at a remote location can verify the status of an enclave. To support remote attestation, Intel provides the Quoting Enclave and Intel Attestation Service. The Quoting Enclave is a special enclave that can generate a *quote*, which is a signed report of an enclave, by going through the similar process to the local attestation. This quote is signed by the hardware key that is issued by Intel, and Intel Attestation Service provides APIs for verifying a quote for its validity. Thus, after a remote verifier obtained a quote from an enclave, it must submit the quote to IAS to check whether or not the quote is valid.

I/O handling in SGX. Because the design of SGX set the enclave run in the user-space (i.e., ring 3), the enclave cannot directly handle I/O requests, which typically require kernel (i.e., ring 0) privilege [37]. Insteads, the application runs outside the enclave handles I/O request on behalf of the enclave, in cooperation with the operating system in the same way how a regular process works in the OS.

Since the actual I/O is handled by the untrusted operating system without having any protection, the enclave must protect I/O channel by itself to ensure the confidentiality and the integrity of data transmitted through the channel. Thus, Intel recommends to use the remote attestation protocol with Diffie-Hellman key exchange [90, 26, 4] to establish a secure communication channel between an enclave instance and a remote host, and to use the *sealing* feature [4] that provides authenticated encryption using hardware-based key to store data generated by an enclave permanently on the disk. Additionally, many other research projects that rely on Intel SGX for the confidentiality and the integrity protection of data utilize Transportation Layer Security (TLS) to provide authenticated encryption to its network communication channel [16, 94, 143, 77, 11, 136].

User I/O in SGX. Unlike network connections that can negotiate security parameters over the channel, user I/O devices such as keyboard and mouse, which are dumb I/O devices, cannot negotiate encryption scheme for establishing a secure communication channel with an enclave. The other general I/O devices are the same, for example, graphic display devices (i.e., a GPU), which requires direct memory access (DMA) for processing data, cannot work with an enclave securely because they do not have a protocol for establishing a secure communication channel.

Intel provides a trusted output path for audio and video outputs through the protected audio and video path (PAVP) [79] on the chipset, which encrypts data that is being transmitted in the bus channel. However, its protocol is proprietary and only the Intel HD Graphics device, which is an integrated GPU to the processor, can support the protocol. Therefore, regular I/O devices cannot make use of this protocol.

5.2.2 Related Work

Comparing SGX with other TEEs. Prior research projects have developed various TEEs in both software and hardware platform. For the software-based TEE, security hypervisors backed by trusted platform module (TPM) such as Flicker [113], TrustVisor [112], and

more generalized work the Extensible and Modular Hypervisor Framework (XMHF) [155] provides a trusted hypervisor that can provide a strong guarantee of isolation on the system. Additionally, seL4, a small, verifiable microkernel that guarantees the correctness of the system thus it protects and isolates a process in a provable manner. However, these software-based TEEs still requires trusting a set of privileged software and also limited in supporting full-fledged applications in their TEE. In contrast, Intel SGX does not trust any privileged software. Moreover, Intel SGX can run unmodified large applications such as Apache, GCC, and the R interpreter [152].

For hardware-based TEEs, the ARM architecture introduced TrustZone and AMD recently added a new feature called secure encrypted virtualization (SEV). Unlike Intel SGX, ARM TrustZone has a trusted path for I/O. The threat model of TrustZone requires a separated, trusted operating system, and this operating system handles I/O request for the applications run on this OS in the secure domain. However, this design requires to include secure operating system into the trusted computing base (TCB), which will enlarge the attack surface of the TEE.

The AMD Ryzen processor also provides the TEE through its new feature called secure encrypted virtualization (SEV). The threat model of SEV is similar to, but a little differs from SGX. In particular, while SGX does not trust underlying OS but runs an application in the enclave in ring 3, SEV does not trust underlying hypervisor but runs the entire virtual machine in ring 0. Although the virtual machine instance of SEV runs in privileged level, the direct memory access (DMA) area for I/O remains unencrypted because I/O devices cannot handle the encryption.

Trusted I/O Paths. Zhou et al. [168, 169] have built trusted I/O path in commodity a commodity x86 system using a small trusted hypervisor and driver. In the first version of the work [168], they isolated a PCI device from an untrusted OS by using the trusted hypervisor. However, the design cannot create an end-to-end trusted path from the user to the application because the application is unprotected and runs outside the TCB. In the second version of the

work [169], they enabled on-demand, isolated trusted I/O path. On isolating the execution environment, their approach requires running of a trusted hypervisor, a small, trusted *wimpy* kernel and a wimpy application on top of the trusted kernel. In contrast, SGX-USB utilizes Intel SGX as a hardware-based isolation mechanism and does not require trusted hypervisor or trusted kernel.

Li et al. [104] have built a trusted I/O path on ARM TrustZone while running device driver in the insecure domain. However, their protection mechanism relies on the randomization of colors on the screen or the randomization of the keyboard layout, all of which are not directly serving I/O to the TEE nor can support general I/O devices.

Martignoni et al. [110] built a trusted terminal for the applications in the cloud. While their thin-client implementation looks promising with a small TCB, however, this method can only be applied to applications running on the cloud machine.

5.3 Overview

SGX-USB aims to establish a trusted communication channel that supports user I/O devices to an enclave instance of Intel SGX. In particular, SGX-USB opens a secure channel that can support USB devices because USB can support a variety of user I/O devices including a keyboard, mouse, camera, speaker, and display, and even for non-user-facing devices such as a disk. Unfortunately, regular USB devices do not have a capability for negotiating and applying security parameters to its I/O channel.

To resolve this, SGX-USB places a proxy device that sits in the middle of the channel and establishes a secure communication channel between an I/O device and the enclave. Establishing a secure communication channel starts with the remote attestation process that authenticates an enclave and the proxy device and shares a secret between these two at the same time. After authenticating and sharing a secret, the proxy device opens a communication channel between a USB I/O device and an enclave and protect the data transmitted in the channel by using a derived encryption key from the shared secret.

The remote attestation process ensures the authenticity of both channel end points, and applying authenticated encryption ensures that all I/O requests to/from the target USB device will securely be delivered through the channel; in other words, no system component other than the enclave can access the I/O data.

5.3.1 Security Guarantees

SGX-USB provides the following security guarantees on a secure communication channel that it establishes between an enclave and the USB Proxy Device.

- **Authenticity**: SGX-USB establishes a secure channel only if it can verify its end points: an enclave and the USB Proxy Device. The I/O request on the channel will be encrypted with a secret key that is only shared between an authenticated enclave and an authenticated proxy device.
- **Confidentiality**: SGX-USB encrypts all I/O request that flows on the channel using an encryption key that is secure derived from a shared secret. Because the remote attestation process ensures that no system components other than two end points of communication channel will get the knowledge of encryption key, no attackers can obtain plaintext data of I/O requests.
- **Integrity**: the encryption also guarantees the integrity of I/O request that flows on the channel. Because SGX-USB uses an authenticated encryption scheme that is resistant to data modification, the replay attack, and the reordering attack, no attacker can inject I/O request on the channel.

5.3.2 Threat Model

We make the following assumption on modeling the threat on building SGX-USB to provide a secure I/O channel from a USB device to an SGX enclave.

- The operating system running on the computer that runs enclaves is untrusted. This assumption assumes that attacker can compromise the entire software stack except for the program in an enclave, including applications running on the OS (not an enclave

application), libraries, drivers, and the kernel.

- We trust the hardware component of Intel SGX (i.e., the processor) and the remote attestation service (Intel Attestation Service, IAS) provided by Intel. This assumption let us verify the integrity of the program that runs inside enclave and share secret keys with an enclave and the USB Proxy Device with authentication.
- We trust the software stack that consists the USB Proxy Device (UPD). The trusted stack includes the operating system kernel of UPD, the `usbip` driver [124], system libraries, and the proxy application runs on the device.
- We assume attackers cannot have the physical access to any of the machine components that consists SGX-USB. Moreover, we assume that the hardware devices that we use for building SGX-USB are trusted. This assumption means that the attacker does not have the power to access the USB device directly (e.g., the keyboard) or to implant a hardware backdoor.

The assumption of trusting the components of Intel SGX (both hardware and software) and not trusting operating system conforms to the typical threat model of Intel SGX. Although we implement the UPD with commodity OS (i.e., `Ubuntu 16.04 LTS`), which contains a large code base, the reason behind this decision is for fast prototyping. We believe that implementing the UPD with minimal TCB is possible; for example, a UPD can be implemented by using small trusted hypervisor such as XMHF or using the other TEEs such as TrustZone; or implemented entirely in hardware. We further discuss on this in §5.8.

5.4 Design of SGX-USB

5.4.1 Architecture

To provide the required security properties to the channel that is established to an enclave, the SGX-USB system consists of following components: an enclave program, the Remote Attestation Service Provider (RASP), and the USB Proxy Device (UPD). Figure 25 illustrates how the SGX-USB components are connected.

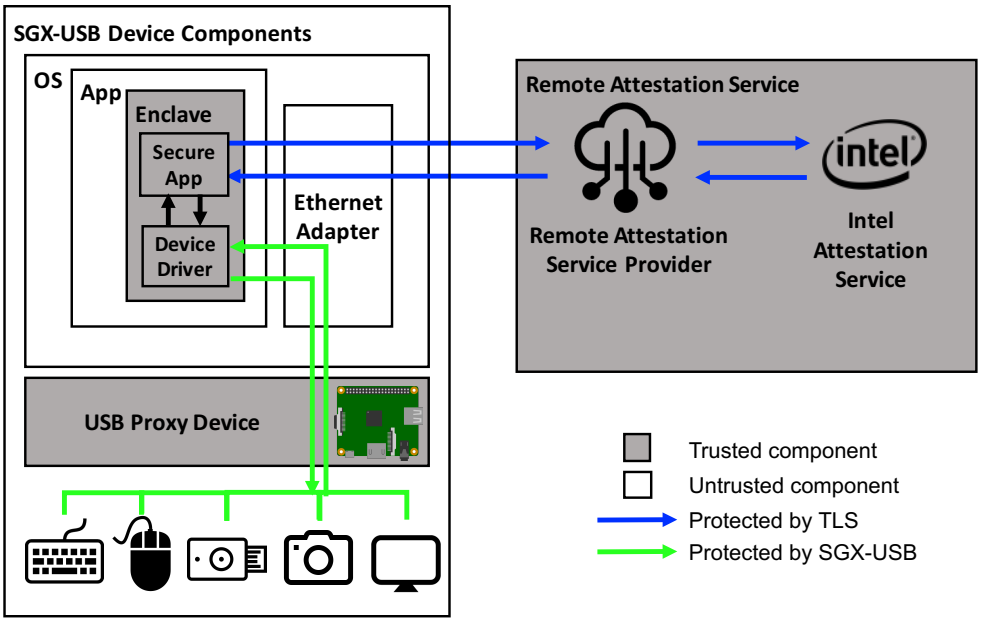

Figure 25: A diagram that illustrates the architecture of SGX-USB. An application that handles I/O runs in the enclave. The enclave will authenticate with the remote attestation service provider (RASP) through the Intel SGX remote attestation process. Intel Attestation Service (IAS) will provide the verification of a quote generated for an enclave, to verify the authenticity of an enclave. The USB Proxy Device (UPD) will receive the signed quote then verifies the signatures of the quote, and then establishes a secure communication channel with the enclave and forward USB I/O devices.

Enclave Program. In SGX-USB, the program that will process I/O must be run in an SGX enclave. This program can be any application that utilizes the secure I/O channel. For example, on utilizing a secure I/O channel as a secure method of processing password, a program that handles the authentication process with user's password will be running in the enclave.

Because an enclave of SGX cannot directly handle I/O requests, the enclave communicates through the untrusted part of the program (i.e. `ocall`) that handles (untrusted) I/O requests such as networking and exchanging unencrypted traffics with the USB Proxy Device. Over the untrusted channel, an enclave and the USB Proxy Device wrap the channel with an encryption layer to provide security guarantees on the confidentiality and the integrity of the data that they stream through the channel.

To share an encryption key and to verify the authenticity of the channel end point, SGX-USB utilizes the remote attestation process provided by Intel (through Intel IAS)

to prove its authenticity and integrity of the program in the enclave and verifying the USB Proxy Device. This process is handled by the remote attestation service provider (RASP).

Remote Attestation Service Provider (RASP). The RASP handles the verification of the authenticity and integrity of an enclave program through the remote attestation protocol of Intel SGX. The RASP is a server program that resides on the network (i.e., on the Internet) and verifies whether or not the current enclave program is intact. By communicating with the Intel Attestation Service (IAS) and the enclave, the RASP receives a quote that is generated by the enclave, which indicates the launching status of the enclave, and sends the quote to the IAS to get a signed quote. Subsequently, the RASP signs the quote by its private key to make sure the authenticity of the ECDHE security parameter in the quote, which will be used for establishing a secure communication channel between the USB Proxy Device and the enclave.

Intel Attestation Service (IAS). Intel Attestation Service is a part of the remote attestation infrastructure of Intel SGX. The job of the IAS is to verify a quote generated by the Quoting Enclave, which is a signed data of a measurement report of an enclave. Because all the quotes of enclaves are protected by a secret key that is fused in the processor and only Intel knows, only the IAS can verify the legitimacy of the quote. The RASP verifies the quote received from an enclave using the IAS to ensure the authenticity of the enclave.

USB Proxy Device (UPD). The USB Proxy Device is a proxy that forwards packets from USB I/O devices to an enclave through secure communication channels. The UPD sits between USB I/O devices and the enclave, and it acts as a middle man that creates secure I/O channel and forwards the I/O requests. To establish the secure channel, the UPD first shares a secret with the enclave program by following the remote attestation process. After sharing a secret, the UPD derives an encryption key and apply an encryption layer to the channel between an enclave and itself to make the channel secure. After establishing a secure communication channel protected by encryption, the UPD forwards USB packets

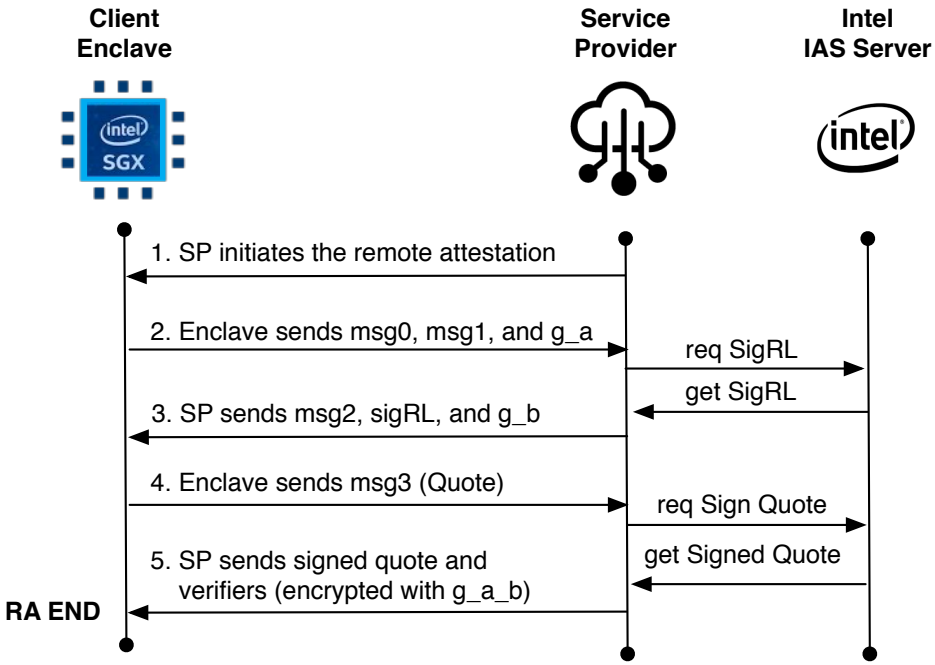

Figure 26: The remote attestation process of Intel SGX.

from the target USB device to an enclave, and from an enclave to the target device, *vice versa*.

5.4.2 Verifying Authenticity and Sharing Secret through Remote Attestation

Before establishing a secure communication channel between the UPD and an enclave, both components authenticate each other to check if the each end point of the channel is intact. Because the regular remote attestation protocol provided by Intel only allows us to verify an enclave from the RASP, we extended the protocol to let the UPD verify an enclave and sharing a secret between them.

Intel SGX Remote Attestation. Intel SGX provides a way of attesting the launching status of an enclave through the remote attestation protocol. Figure 26 illustrates how this process works. In the following, we describe each step of the protocol.

1. The service provider (the RASP), which is a remote party that requests the verification of an enclave, initiates the remote attestation process.

2. The enclave that is being attested gets the request then send `msg0`, which contains group ID of an enclave and `msg1`, which contains the public key (i.e., `g_a`) parameter of the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) protocol that will be used for sharing a secret with the service provider at the end of the remote attestation process.

3. Next, on receiving both `msg0` and `msg1`, the service provider verifies the group ID (must be 0) in the `msg0`. If the group ID is zero, then the service provider requests a revocation list (i.e., `SigRL`) from the Intel IAS. This `SigRL` is signed by Intel IAS and will be used by the enclave for verifying the validity of the service provider. After processing the messages, the service provider generates `msg2`, which contains its public key parameter for the ECDHE key exchange (i.e., `g_b`).

4. After receiving `msg2`, the enclave generates a report and gets a quote for the report by the Quoting Enclave (QE). A report of an enclave includes the measurements (i.e., hash) of the launching status of an enclave that only Intel Attestation Service can verify as well as both of public key parameters (`g_a` and `g_b`) for the ECHDE key exchange. The Quoting Enclave, an enclave that is developed by Intel, will sign the report, and the enclave sends this quote to the service provider as `msg3`.

5. Subsequently, the service provider receives `msg3` and send it to the IAS to verify whether the quote is valid or not. Only for the valid quote, the IAS will return a signed quote with a signature generated by Intel's private key. The service provider verifies this signature; if it is valid, the service provider generates a shared secret and then send the signed quote to the enclave as `msg4`, by encrypting the signed quote with the secret key.

6. The enclave also calculates a shared secret and derives an encryption key; then it decrypts the quote from `msg4` using the key and verifies the signature of the quote. In consequence, the enclave can ensure that it has shared a secret with a service provider

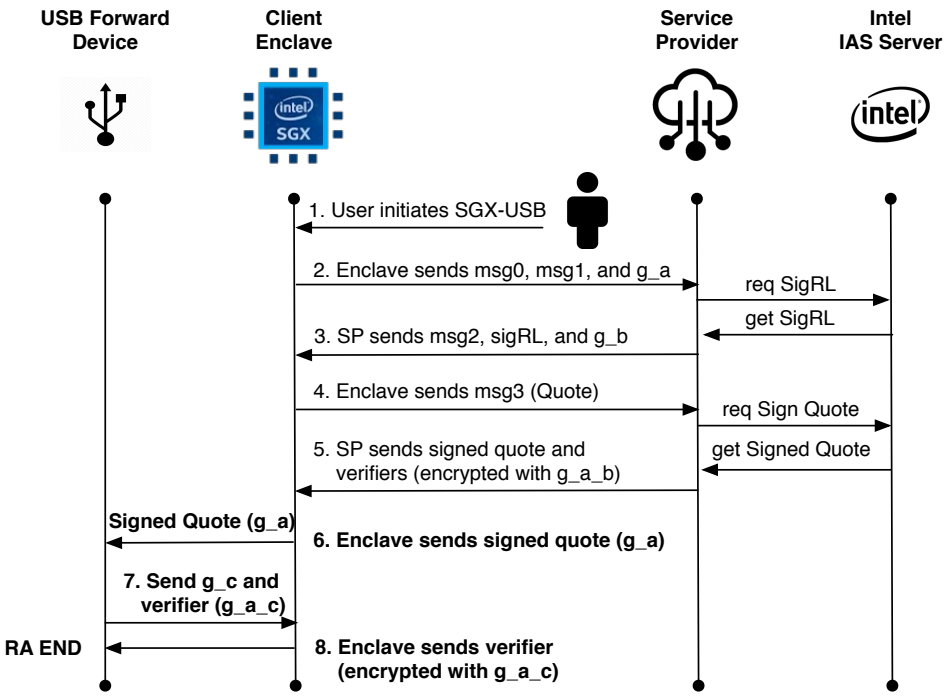

Figure 27: An extended remote attestation process for SGX-USB. Steps from 1 to 5 remain the same as the regular remote attestation of an enclave. Procedures marked with the bold face (Steps 6, 7, and 8) indicate additional procedures for attesting an enclave from the USB Forwarding Device.

that is certified by Intel (because they cannot get the correct signature from IAS unless Intel does not certify them), and the service provider can ensure that it has shared a secret with a legitimate enclave instance (because IAS will not sign the quote if an enclave instance is not legitimate). Note that both the enclave and the RASP have shared a secret (i.e., `g_a_b`) through the ECHDE protocol.

SGX-USB Remote Attestation. To share a secret between the UPD and an enclave, we extended the remote attestation process of Intel SGX. Figure 27 shows the process of the remote attestation with our extension. We describe the extended part of the process in the following.

- For the step 1, we changed the process to be user initiated instead of the service provider.
- After the step 5, the enclave sends the signed quote (decrypted from `msg4`) to the UPD.

- On receiving the signed quote, the UPD verifies if the quote is correctly signed by the Intel's private key and the private key of the RASP. Only if both signatures are verified, the UPD generates an ECDHE parameter and send the public part (i.e., `g_c`) to the enclave as `msg5`, along with the signature of this public parameter and the public key that can verify the signature. Note that the UPD presents a public key as a certificate that contains its signature, signed by the RASP.
- On receiving `msg5` from the UPD, the enclave verifies the signature of the ECDHE parameter (i.e., `g_c`) using the public key of the RASP, to check if the RASP has certified the signing key. As a result of the process, the UPD has verified that both the IAS and the RASP have signed the quote (so it is valid), and the enclave has verified that the signature of the public key parameter (i.e., `g_c`) is generated by a public key that is certified by the RASP (so the UPD is certified one). Only if all signatures are verified, both the enclave and the UPD calculates a shared secret (i.e., `g_a_c`) and derives an encryption key that will be used for securing the I/O channel.

5.4.3 Trust Chain and User Verification

Trust Chain. For guaranteeing the authenticity of the channel end points, SGX-USB trusts two private keys during the remote attestation process. First, we trust the private key of Intel Attestation Service (IAS) that is used for signing quotes of an enclave. The quote from an enclave, an evidence of running a correct enclave, can only be verified by Intel IAS. By trusting Intel IAS, the USB Proxy Device can verify that it currently communicates with a real, valid enclave instance.

Second, we trust the private key of the remote attestation service provider that signs the public key of the USB Proxy Device. In this regard, an enclave can verify the validity of the UPD by checking if the public key of the UPD that signs its ECDHE parameter is issued by the RASP, by verifying its signature.

User Verification. Although the public key infrastructure and cryptographic operations can

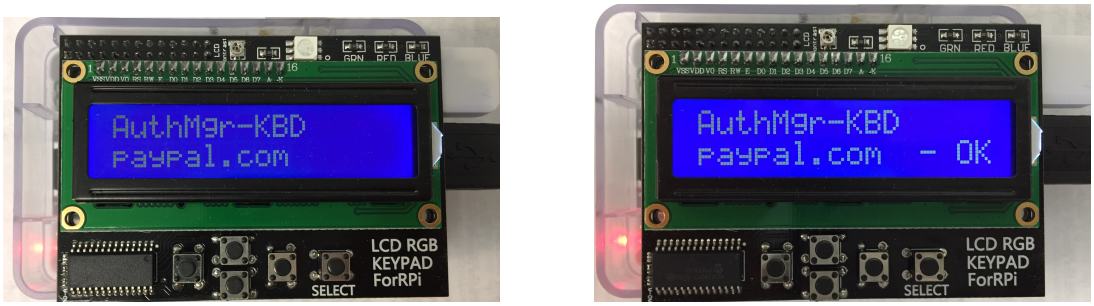

Figure 28: The user interface for verifying an enclave and its usage, presented in the USB Proxy Device. Figure on the left shows how the UPD displays the request for establishing a secure channel to a keyboard from an enclave. The information displayed on the LCD screen indicates the name of an enclave (i.e., AuthMgr), the name of the requested device (i.e., Keyboard), and application specific information for indicating the usage of the input (i.e., `paypal.com`). After clicking the `SELECT` button (i.e., the user approves), the screen will show the 'OK' sign at the end of the second line to indicate that the secure channel is established.

guarantee the authenticity, the confidentiality, and the integrity of communication channel, the establishment of the channel must go through the user verification process to ensure that the use of the channel follows the user's intent.

To this end, the UPD explicitly display the identity of an enclave (e.g., application name), the device that the enclave connects to (e.g., keyboard), and the usage of the device (e.g., that domain name that the keystrokes will be submitted).

Figure 28 shows an example of the user verification process of SGX-USB on using the system as a password authentication manager. On the screen, the first line displays the application name as *AuthMgr* and the device name as *KBD* to indicate that the AuthMgr enclave would like to talk to the keyboard device. In the second line, the UPD will display how the user input will be used for, in other words, displays the domain name `paypal.com` to indicate that the password typed by the user will be submitted to the `paypal.com`.

To authorize the access, the user can click 'OK' button on the device (indicated as *Select* in Figure 28). The channel will be established only if there is a user approval; otherwise, the UPD will not make the connection to the device.

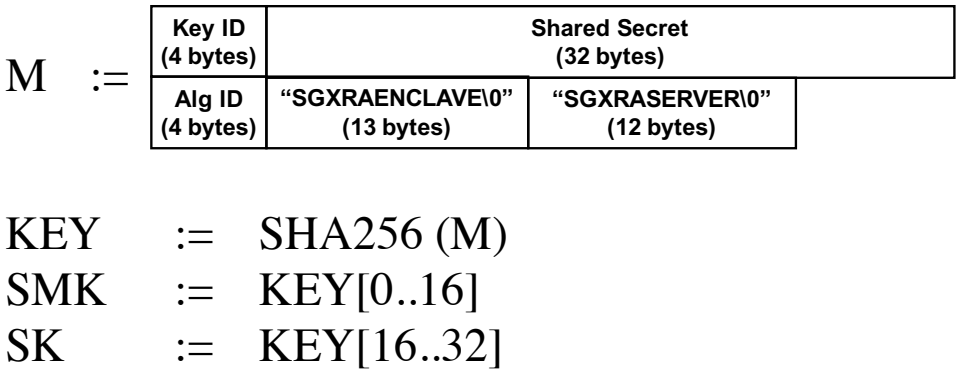

Figure 29: The data format for deriving secret key from a shared secret. The key derivation function uses the SHA-256 message digest algorithm to derive a 16 bytes secret key from a shared secret.

5.4.4 Integrity and Confidentiality: Encrypted Communication Channel

To protect the communication channel between an enclave and the UPD, SGX-USB wraps the channel with an encryption layer protected by the key that is exchanged during the remote attestation process. In short, SGX-USB applies the AES-128-GCM scheme, which is an authenticated encryption with associated data (AEAD) that can protect both data confidentiality and data integrity.

Key derivation. After finishing the remote attestation process, both an enclave and the UPD have shared a 256-bit secret through ECDHE protocol using the NIST P-256 curve. To derive a 128 bit key for an AES encryption, SGX-USB followed the same way on how Intel derives a secret key in their SDK example; the scheme uses the SHA-256 message digest algorithm. Figure 29 illustrates how the key derivation function works. By hashing the Key ID (0 in this case), the 32 bytes shared secret, the Algorithm ID (0 in this case), and two string literals `SGXRAENCLAVE` and `SGXRASERVER`, the derivation function generates a 32 bytes message digest and uses the latter 16 bytes (*SK* in Figure 29 for the encryption key.

Encryption Scheme. SGX-USB uses `AES-128-GCM` for the encryption scheme for the secure channel. Since the GCM (Galois Counter Mode) is an authenticated encryption with associated data (AEAD) encryption scheme, we can use one key for protecting both the confidentiality and the integrity of the data. To encapsulate a plaintext USB packet into an

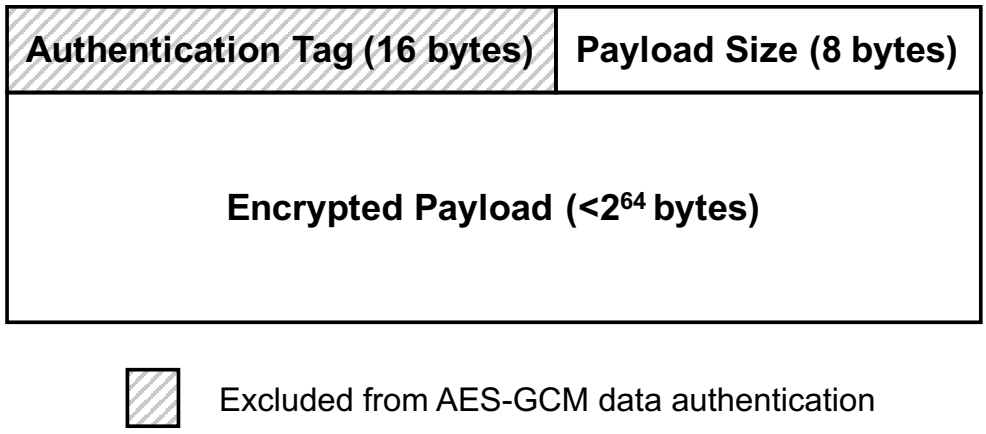

Figure 30: The header format for delivering encrypted payload on trusted I/O channel in SGX-USB. Authentication Tag will be used for verifying the integrity of both the size field and encrypted payload. While the AES-128-GCM encryption applied only to the payload, the size field is supplied as additional data for AES-128-GCM data authentication; thus the encryption scheme protects the integrity of both encrypted payload and the size field.

encrypted packet, we attach a 24 bytes header on the payload followed by the encrypted packet payload. Figure 30 shows how the header of a packet in the channel composed.

To send a plaintext USB packet over the secure channel, we first identify the size of the packet. To protect the integrity of both encrypted text and the size field, we encrypt the packet payload using the AES-128-GCM encryption scheme. At the same time when the encryption is being processed, we put the size (8 bytes) field as the additional data to be authenticated. In this way, the scheme allows us to detect any forgery on both encrypted data and the size field. As a result of an encryption routine, the scheme will generate a 16 bytes authentication tag that will be verified when decrypting the data to check if the data is intact. We put this tag at the top of the header to deliver the tag to the other end point.

We process the decryption in a reverse way. After receiving the header data, we initialize a decryption engine with the secret key, the authentication tag in the header, and the size field in the header as the additional data to authenticate. The encryption scheme will return true only if when the secret key and the authentication tag is matched.

Another point that is essential on applying the AES-128-GCM encryption scheme is the setting of initialization vector (IV). To securely use the encryption scheme, one must not

reuse the IV for one secret key. To follow such a secure scheme, we set a 12 byte (96bit) integer counter value starting with zero value for each of sending and receiving side, and increment IV counter per each en(de)cryption operation. In this regard, the IV will not be reused if the channel sends less than 2^{96} packets, which is a practically unreachable number. Additionally, because the IV counter is monotonically increasing on each sending and receiving side, the scheme is resistant to the replay and the reordering attack.

5.5 Use Cases

This section illustrates potential use cases of SGX-USB. We present examples for applying SGX-USB on protecting user's password, and on establishing end-to-end trusted I/O channel for video chat. Although we only cover these two use cases as examples, theoretically, SGX-USB can support any USB device since SGX-USB forward USB I/O channel through established secure communication channel.

5.5.1 AuthMgr: Protecting User's Password using SGX-USB

Password is an essential user input that requires confidentiality protection because it is a credential for a *What-you-know* factor of the authentication. Attackers can impersonate themselves as a legitimate user if an attacker possesses user's password. Under typical operating system settings, there are several ways of attacking the password input. For attackers with the kernel level privilege, they can easily intercept USB HID messages to record keystrokes that user types on a password box. Even with the lower, user-level privilege, attackers can either directly read process memory or build user-level keylogger to obtain the typed password.

To block such attack pathways, we apply SGX-USB to protect user's password. Suppose a user wants to log on to Paypal while his/her machine has already been infected by a kernel level malware (assuming the highest threat). When the user types a password, the user's password can be recorded or stolen from the memory by methods mentioned above.

However, SGX-USB can block such attacks. Before the user types a password, SGX-USB

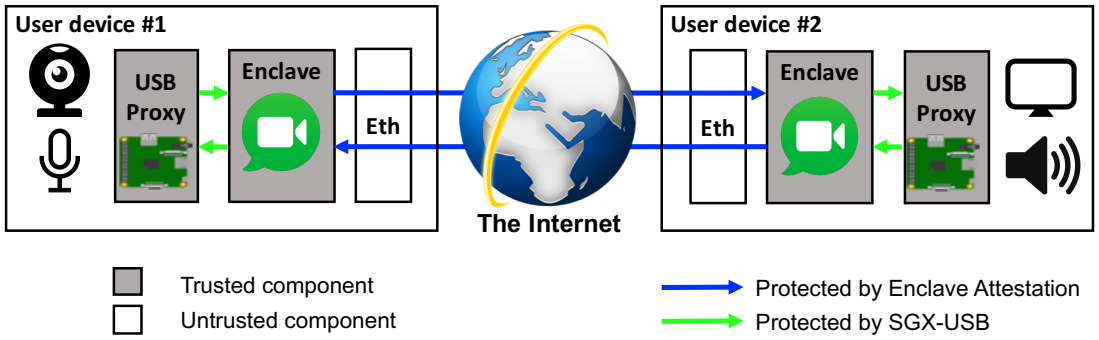

Figure 31: A diagram that illustrates the end-to-end I/O protection use case of SGX-USB for the Internet video chatting. The USB proxy device on the user's machine will forward USB devices required for video chatting such as camera, microphone, speaker, and display. The video chat application running in the enclave can securely access these USB devices, and send I/O data through the secure communication channel over the Internet between the enclaves.

will establish a secure communication channel to an enclave that handles the authentication process with the password. After the user identifies that the secure channel is established, the user types his/her password, and user's keystrokes will be available only to an enclave as plaintext. Because the secure communication channel ensures that the attackers cannot harm the confidentiality of the data, the attackers will have no knowledge about the password. Additionally, Intel SGX guarantees that no program other than the enclave, including the kernel malware, can read the data in the enclave's memory; thus the password is protected from the attack.

After getting the password from the user, the enclave handles the authentication process to the service using a secure network channel such as HTTPS, which is protected by TLS. When the service accepts the password and returns a session cookie, the enclave will only return the cookie as a one-time session token, which will become invalid after the user logs out after he/she finishes the session, to the user's web browser.

Although the malware could steal the session cookie from the machine, the password remains secret to the attacker.

5.5.2 Internet Video Chatting: A Potential Use Case

SGX-USB can establish not only a trusted I/O channel to an enclave within the machine but also an end-to-end trusted I/O channel between an enclave in the machine and enclaves over the Internet. Figure 31 illustrates how the end-to-end trusted I/O channel can work with an example of video chat over the Internet.

On one end device, the UPD establishes multiple secure communication channels to the video chat application running in an enclave, and then forwards camera and microphone devices to the enclave. On the other end device(s), the UPD of a remote machine establishes secure communication channels to the video chat application running in an enclave, and then forward display and speaker devices to the enclave. Finally, the enclave on one end and the enclave on the other end establishes a secure communication channel (through the authentication method in the video chat service). After all the connections are established, the video and audio stream data provided by a user at one end can securely be delivered all the way down to the other end's display and speaker device.

Because all channels are established with proper authentication (by the remote attestation process of SGX-USB and the authentication in the video chat application) and protected with confidentiality and integrity guarantee, any attacker in the middle including OS, untrusted application, and network and service operators can neither hijack the communication nor evaesdrop the channel. Moreover, the video chat application is protected by an enclave of Intel SGX; thus no software attacker can eavesdrop or altering video chat data from memory or other system resources.

5.6 Implementation

We implemented our prototype of SGX-USB using a desktop machine that supports Intel SGX and Raspberry Pi 3 model B device, which is a small board computer, and securely forwarded a keyboard device to an enclave to reflect the AuthMgr use case in §5.5. In the following, we describe the details of implementation settings.

Table 8: Source code line count for the software components of SGX-USB.

Component	Module	Language	SLOC
Enclave	Core	C++	571
	USB HID Driver (Keyboard)	C++	140
	OCALL Layer	C++	125
USB Proxy Device	Remote attestation and Proxying	C++	658
	LCD Controller	Python	32
Remote Attestation Service Provider	Server	C++	280
Common Library	Crypto, Socket, etc.	C++	2,944

Enclave. We implemented the enclave application that securely processes a user's password for login (AuthMgr in §5.5) on a desktop machine equipped with a quad-core Intel Core i7 6700K (4.0Ghz) processor, which supports Intel SGX, attached with 32GB of DDR4 RAM and running Ubuntu 16.04.2 LTS. For the software component for the enclave, we used the Linux version of the Intel SGX SDK (v1.9), which we can clone the code publicly from Github. The trusted part of the program consists of 571 lines of C++ code for handling the establishment of a secure channel including remote attestation and encryption, and 140 lines of C++ code for handling keyboard input through the USB HID protocol. The `ocall` layer, which is untrusted part of the program, for transmitting network packet to the UPD and the target login service through TLS is composed of 125 lines of C++ code.

USB Proxy Device. We implement the USB Proxy Devices using a Raspberry Pi 3 Model B device equipped with a quad-core ARM Cortex-A53 (1.2Ghz) processor, attached with 1GB DDR3 RAM and running Ubuntu 16.04.2 LTS for the `armv7l` architecture. We construct the communication channel between the desktop machine and the USB Proxy Device using a Gigabit Ethernet adapter connected to the USB 2.0 port of Raspberry Pi, which supports around 310Mbps for its maximum throughput. For proxying USB devices to the network adapter, we use the USBIP [124] project which is included in the Linux kernel. For the part that handles the establishment of a secure channel including the remote attestation process

and encryption of packets, we composed it with 658 lines of C++ code. For controlling the LCD display on the USB Proxy Device, we implemented a Python program that uses `AdaFruit_CharLCD` library with 32 lines of code.

Remote Attestation Service Provider. We implemented the remote attestation service provider (RASP) for handling the Intel SGX remote attestation process using OpenSSL and `libcurl` for communicating with the Intel IAS, on a server equipped with Intel Xeon E3-1271 v3 (3.6Ghz).

5.7 Evaluations

We evaluate SGX-USB by answering the following questions:

- How secure is the I/O communication channel established by SGX-USB? (§5.7.1)
- How much overhead does SGX-USB incur on delivering I/O packets in terms of throughput and latency? (§5.7.2)
- How long does it take to establish the secure I/O channel through the remote attestation process? (§5.7.2)

5.7.1 Security

The threat model of SGX-USB excludes the operating system from its trusted computing base (TCB). So there are several points in the system that attacker can intervene before and after SGX-USB establishes a secure I/O channel. In the following, we go over attack cases and show how SGX-USB block such attacks.

Attacks against enclave instances. The first avenue for an attack is to thwart the protection provided by an SGX enclave instance. Because Intel SGX isolates all memory access from the entire domain controlled by attackers including operating system, attackers cannot obtain nor alter the runtime data in the enclave's memory. Additionally, attackers cannot change the behavior of an enclave instance because the remote attestation process ensures that the integrity of the code in the enclave. One possible way would be launching a malicious

enclave instance and establishing secure I/O channel using this enclave. However, because the remote attestation process of SGX-USB not only includes the IAS but also utilizes the RASP, the RASP will not generate a signed quote if the enclave instance (and its measurement) is not pre-registered to the service. By only using the set of pre-registered enclave applications, an attacker could launch a disguising attack that invokes a different enclave instance with the enclave what user wants to use. In such a case, at the final verification stage of SGX-USB, the user will notice that the name of the malicious enclave instance is not the enclave that user has his/her intention, so request for establishing secure I/O channel will be rejected.

Attacks on the remote attestation procedure. Attackers could try to forge an acceptable message during SGX-USB's remote attestation process. First, an attacker could attempt to generate a fake quote; for example, the attacker could present a valid quote message with a legitimate measurement for a registered enclave while executing a malicious enclave instance. Nonetheless, this is strictly protected by the Intel SGX hardware and the IAS. All the measurement reports must be generated by the secret key fused into the processor hardware, and the quote can only be verified by the Quoting Enclave, which is created by Intel. Thus, the attacker cannot either generate valid quote with forged message or pass the verification process of the IAS.

Another avenue of attacking the remote attestation process is to forge the message generated by the USB Proxy device. Again, forging message requires generating the correct signature that bounds to the private key of the USB Proxy Device or a private key signed by the RASP. Without obtaining the private key of these trusted instances, no attacker can forge the message on the remote attestation process.

Finally, an attacker could attempt to build a fake USB Proxy Device to inject arbitrary I/O message to an enclave instance. Although we set our threat model to exclude attackers with any physical access to the device, the attacker could obtain an instance of the

USB Proxy Device if the device available in public. Because current prototype implementation builds the USB Proxy Device as a small computer instance, the attacker who can obtain the device can disassemble to leak the private key signed by the RASP and use the key to build a fake instance of the USB Proxy Device. However, we believe that this is just an implementation issue; this can be protected by implementing the UPD with the other TEE or entirely in hardware. We further discuss on other trusted ways of building the USB Proxy Device in §5.8.

Attacks on the secure channel. Attackers could attempt to decrypt or inject data on the established secure channel. Unfortunately, the secure channel is protected by a symmetric encryption scheme, `AES-128-GCM`, which is an authenticated encryption with associated data (AEAD). The correct use of the scheme guarantees that no attackers can decrypt or alter the encrypted data without obtaining the encryption key. To achieve this guarantee, SGX-USB follows the same way in how Transportation Layer Security (TLS) utilizes the same scheme as AEAD (e.g., use decrypted data only if tag matches, encrypt only short block, does not reuse the same IV, etc.). Moreover, because ECDHE key exchange scheme securely derives the key, attackers can obtain the key only if by breaking the scheme or by forging the key exchange message, all of which are impossible in SGX-USB construction.

Despite the fact that SGX-USB can guarantee the authenticity of channel end points and the integrity and the confidentiality of the data on the channel, SGX-USB cannot guarantee the availability of the channel. We further discuss on this limitation in §5.8.

5.7.2 Performance

We evaluated SGX-USB for the performance of I/O channel in terms of throughput and latency. Moreover, we provide timing information how long does the establishment of a secure I/O channel through remote attestation takes.

Throughput and Latency. To evaluate the performance of the secure I/O channel established by SGX-USB, we measure the throughput and latency of the channel for various

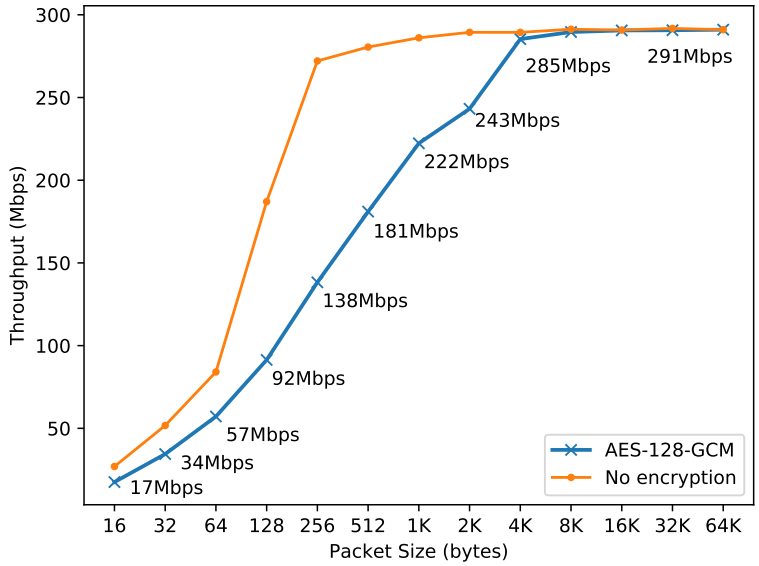

Figure 32: The measured throughput of the secure I/O channel for various packet size.

packet sizes. To help understand the result, we note that the maximum packet size of the USB protocol is 1 KB and typical packet size for USB HID devices is 32–512 bytes. Although we use a Gigabit Ethernet adapter (max bandwidth 1Gbps) for the communication channel, the adapter is connected to a USB 2.0 port (max bandwidth 480Mbps) due to the hardware limitation on the USB Proxy Device so the maximum bandwidth of the channel is around 310Mbps without any encryption or encapsulation.

Figure 32 depicts the throughput of the secure I/O channel for various packet size and Table 9 lists detailed numbers.

The smaller packet size incurs more overhead on both encryption process (CPU) and size (bandwidth). Because SGX-USB applies a separate instance of `AES-128-GCM` encryption (i.e., using a different IV) per each packet, the number of required encryption initialization process is increased for the smaller packet size. Moreover, because SGX-USB adds small header data (24 bytes) per each packet for transmitting data authentication tag (16 bytes) and indicating payload size (8 bytes), delivering smaller packet would incur more size overhead. Furthermore, we deliver USB packets over a TCP connection on the Ethernet link,

Table 9: The measured throughput of the secure I/O channel for various packet size, in five seconds of transmission. The throughput measured by the amount of payload data transmitted on the channel without counting any additional data for encapsulation. *W/O encapsulation* indicates the channel throuput when we count the entire amount of data transmitted through the channel including header information. *No encryption* indicates the channel throughput when we applyed payload encapsulation (i.e., adding of the header) but did not apply encryption.

Packet Size (Bytes)	**64**	**128**	**512**	**1024**	**4096**	**8192**
W/O encapsulation (Mbps)	181.3	295.1	309.6	306.8	294.6	292.2
No encryption (Mbps)	84.1	187.0	270.5	286.1	289.4	289.6
AES-128-GCM (Mbps)	57.1	91.5	181.0	222.3	285.3	289.4
Overhead (%)	-32.1%	-51.1%	-35.5%	-22.3%	-1.4%	-0.07%

Table 10: The measured average latency of the secure I/O channel for various packet size, in five seconds of transmission. *No encryption* indicates the latency incurred when we applyed payload encapsulation (i.e., adding of the header) but did not apply encryption.

Packet Size (Bytes)	**64**	**128**	**256**	**512**	**1024**	**4096**	**8192**
No encryption (usec)	1.74	1.93	2.39	4.64	9.10	35.99	71.53
AES-128-GCM (usec)	2.85	3.51	4.71	7.19	11.71	36.51	71.94
Overhead (usec)	+1.11	+1.57	+2.32	+2.55	+2.61	+0.52	+0.41

so additional 50 bytes size overhead is applied per each 1426 byte payload (1426 = 1500 (MTU) - 50 (TCP/Ethernet) - 24 (header)).

Because of these overhead characteristics, SGX-USB demonstrated 57.1 Mbps of throughput for 64 bytes packets, which is around 18% of the maximum throughput. However, for 4K bytes packets, the bandwidth became saturated, and the overhead is negligible.

Encapsulating the packet and applying encryption on the packet also incurs overhead on the channel latency. Figure 33 depicts the throughput of the secure I/O channel for various packet size and Table 10 lists detailed numbers. Although the latency increases as the packet size increases, the absolute value of the latency in maximum USB packet size (i.e., 1 KB) is around 11 microsecond, which is fairly negligible.

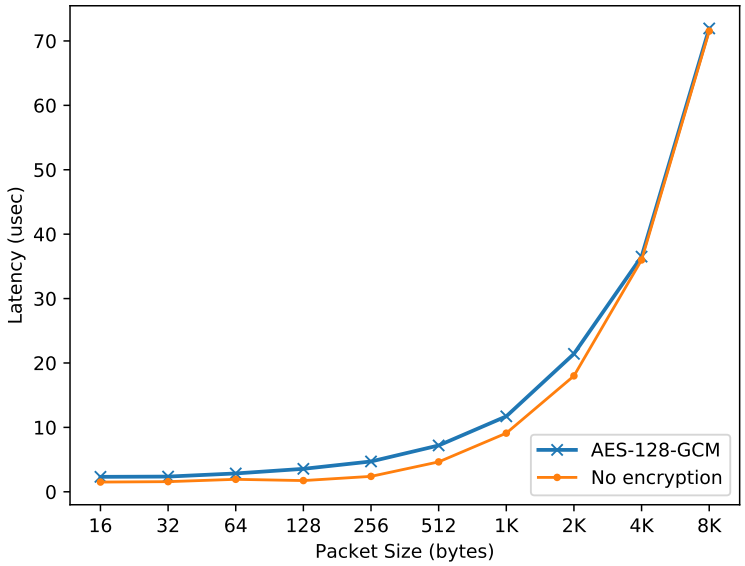

Figure 33: The measured average latency of the secure I/O channel for various packet size, in five seconds of transmission.

Authentication speed. The remote attestation process of SGX-USB requires:

- Two round trips between the enclave and the RASP for delivering `msg0`, `msg1`, and `msg2`; and `msg3` and `msg4`,
- Two round trips between the RASP and the IAS, one for requesting and receiving `SigRL` and the other for and the signed quote,
- One round trip between the enclave and the UPD for exchanging ECHDE parameter (using the signed quote).

To model a realistic use case, we setup the connection between the enclave and the UPD as a local network connection, place the RASP on the remote network using the Google Cloud Platform and using the test IAS server provided by Intel.

The total round trip time for the remote attestation for SGX-USB (Figure 27) from step 1 to step 7, it took in average 553 milliseconds, with standard deviation 31ms for 100 times of remote attestation trials. This overhead is not that much because the entire process of remote attestation is one-time cost per each channel; it only happens when the USB Proxy Device

establishes a new secure communication channel with an enclave.

5.8 Discussions

In this section, we discuss on how SGX-USB can support general I/O, on the performance of the channel, on the feasibility of hardware implementation of the UPD, on authenticating the identity of an enclave, and the availability of the channel, which is an unprotected security property on the channel.

General I/O support with SGX-USB. On forwarding a USB device to a network device, the prototype design of SGX-USB borrows the implementation of the `usbip` [124] project that generally supports all kinds of USB devices, so SGX-USB is. Because the USB protocol transmits its data as packets, delivering each USB packet as a packet over the IP can be done by only incurring transformation overhead. Moreover, since we use transmission control protocol (TCP), which is a reliable protocol, for the data transmission, so there will be no missing packet on the other end. Therefore, as long as the driver software can run in the enclave, SGX-USB can support any USB device by forwarding its packet to the enclave.

In addition to USB devices, we believe that SGX-USB can forward devices that support RDMA (remote direct memory access) protocol through established channel by implementing a driver counterpart in the enclave, because by design, data for RDMA can be delivered over the network.

Channel Performance. Performance evaluation result of SGX-USB shows that its latency is in a performant range, but suffers performance bottleneck due to the encryption process. However, the bottleneck can be removed if the processor supports hardware-based encryption engine. Starting from newer ARM processors, processor manufacturers other than Intel try to integrate hardware module that accelerate encryption speed.

Regarding the size overhead of the channel bandwidth, the higher bandwidth would mostly be used by USB at bulk transfer, which sends a large amount of data split in each 1K byte packet. In such a case, SGX-USB can set a buffer to consolidate multiple USB

packets into a large chunk (e.g., merging 16 packets into a 16Kbytes packet) only for the bulk transfer then the overhead will be negligible.

Hardware implementation of the UPD. Although we implemented our prototype of SGX-USB using a Raspberry Pi, which is a small board computer, we believe that implementing the UPD in hardware or other TEE with smaller TCB is feasible. The hardware implementation of UPD may include a USB host controller to receive raw packets from I/O devices, a communication interface to the enclave device, (in any form, e.g. Ethernet or USB OTG guest device), a cryptographic engine that handles the remote attestation process and AES encryption, and a small storage that is loaded with trusted public keys and a firmware that controls the components.

A more flexible design would be utilizing ARM TrustZone. In this case, by implementing the `usbip` driver on a small and secure TEE OS for TrustZone, we can significantly reduce the size of TCB. Moreover, in conjunction with using TPM, we can securely store the code and the private key of UPD with the data sealing feature; so the attackers with a possession of the UPD cannot alter the code nor retrieve the private key of the device.

Availability of the channel. We exclude the availability from the security property that SGX-USB should guarantee for the communication channel. This is an inherent limitation due to the adoption of the threat model of Intel SGX because Intel SGX excludes the operating system, which runs as a higher privilege than an enclave, from the threat model.

Although guaranteeing availability cannot be possible under current threat model, untrusting the operating system, the user will directly be notified at least when the availability issue happens (i.e., the device does not work at all). The operating status of the channel will either be fully working or not and cannot be half-working status because missing any of USB packet will break the encryption status of the secure channel.

CHAPTER VI

CONCLUSION

In this thesis, we analyzed and protected user I/O in commodity computer systems by identifying the three key security properties of user I/O: integrity, confidentiality, and authenticity.

In Chapter §2, we built **GYRUS** to protects the systems from transmitting non-user-intended network traffic by preserving the integrity of user input from the user interface layer to when the input transformed and transmitted to the network device. By protecting the integrity of user input, **GYRUS** can cut-off various attack pathways including malware threats in an attack-agnostic way.

In Chapter §3, we built **M-AEGIS** to protect user's private data sending and receiving on public messaging services by presenting transparent encryption layer between the user and the application user interface. By protecting the confidentiality of user input/output, **M-AEGIS** can provide a true user-to-user encryption to protect user's private data without modifying underlying protocols and applications.

In Chapter §4, we examined the user I/O security of popular operating systems regarding the authenticity of I/O end points. Missing security checks that verify the source and the destination of user I/O for a system's accessibility support breaks an essential security assumption that the input always comes to the user and the output can only be seen by the user. Failure to authenticate the source of an input let attackers inject input to the system so that attackers can take over the system as if they are a legitimate user even when the state-of-the-art security mechanisms fully protect the system. Failure to authenticate the destination of an output let attackers have access to security-sensitive data such as passwords, which endangers a user.

After identifying the three key security properties of user I/O, in Chapter §5, we built a trusted I/O channel that guarantees these security properties on using trusted execution environment. Enabling trusted user I/O path let Intel SGX support user-facing applications such as authentication manager and end-to-end trusted video chat.

In the following, we conclude by discussing several open problems in user I/O protection.

Open problems. On protecting the integrity of user input, **GYRUS** is still limited only to protect text-based user input that only applied with simple transformation. One promising approach to support arbitrarily complex transformation on user input is to apply probabilistically checkable proof or homomorphic encryption. However, such cryptographic approach still suffers massive runtime overhead to be used as a practical solution.

On protecting the confidentiality of user input, **M-AEGIS** requires a developer's manual effort of building the per-TCA logic of an application. Applying machine learning to automatically learn the UI layout of an application then automatically generating per-TCA logic would be an exciting direction. Moreover, to support end-to-end encryption other than text-based user input encounters another challenge. While we could use the Unicode encoding to place encrypted text to the user interface of a normal application without having compatibility issue, encrypting and transmitting image, audio, or other multimedia data requires format preserving encryption even when the application applies a transformation to optimize the size of content (e.g., image/video compression).

On evaluating the accessibility support in popular systems in the **A11Y ATTACK**, we manually analyzed the system and each accessibility component. Building an automated analysis tool for checking whether or not the information flow of user I/O conforms to a system's security policy would be a good direction to resolve such problem. The challenge on this approach is at the point that the input handling routine is processed asynchronously. To analyze the system, one must figure out how to exhaustively examine the point that an input event can be injected and an output event will be generated. Moreover, designing a new UI framework for securely supporting accessibility would be a great next step.

REFERENCES

[1] 107TH CONGRESS, "Uniting and strengthening america by providing appropriate tools required to intercept and obstruct terrorism (usa patriot act) act of 2001," *Public Law 107-56*, 2001.

[2] ACQUISTI, A. and GROSS, R., "Imagined communities: Awareness, information sharing, and privacy on the facebook," in *Privacy enhancing technologies*, pp. 36–58, Springer, 2006.

[3] ALEXA INTERNET, "Alexa - Top Sites in United States." http://www.alexa.com/topsites/countries/US.

[4] ANATI, I., GUERON, S., JOHNSON, S., and SCARLATA, V., "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[5] ANDROID DEVELOPERS, "Accessibility." http://developer.android.com/guide/topics/ui/accessibility/index.html.

[6] ANDROID DEVELOPERS, "Security tips." http://developer.android.com/training/articles/security-tips.html.

[7] ANDROID DEVELOPERS, "UI Testing." http://developer.android.com/tools/testing/testing_ui.html.

[8] ANDROID OPEN SOURCE PROJECT, "Dalvik Technical Information." https://source.android.com/tech/dalvik/index.html.

[9] APPLE, INC., "Accessibility." http://www.apple.com/accessibility/resources/.

[10] APPLE, INC., "The ios environment." https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/TheiOSEnvironment/TheiOSEnvironment.html.

[11] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., and OTHERS, "Scone: Secure linux containers with intel sgx," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA), Nov. 2016.

[12] ARORA, S. and SAFRA, S., "Probabilistic checking of proofs: a new characterization of np," *J. ACM*, vol. 45, pp. 70–122, Jan. 1998.

[13] BADEN, R., BENDER, A., SPRING, N., BHATTACHARJEE, B., and STARIN, D., "Persona: an online social network with user-defined privacy," in *Proceedings of the 20th ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[14] BALDUCCI, F., "Whatsapp is broken, really broken." http://fileperms.org/whatsapp-is-broken-really-broken/.

[15] BATES, D., "New privacy fears as facebook begins selling personal access to companies to boost ailing profits." http://www.dailymail.co.uk/news/article-2212178/New-privacy-row-Facebook-begins-selling-access-users-boost-ailing-profits.html.

[16] BAUMANN, A., PEINADO, M., and HUNT, G., "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Broomfield, Colorado), pp. 267–283, Oct. 2014.

[17] BEATO, F., KOHLWEISS, M., and WOUTERS, K., "Scramble! your social network data," in *Privacy Enhancing Technologies*, pp. 211–225, Springer, 2011.

[18] BEGEMANN, O., "Remote View Controllers in iOS 6." `http://oleb.net/blog/2012/10/remote-view-controllers-in-ios-6/`.

[19] BELLARE, M., BOLDYREVA, A., and O'NEILL, A., "Deterministic and efficiently searchable encryption.," in *CRYPTO* (MENEZES, A., ed.), vol. 4622 of *Lecture Notes in Computer Science*, pp. 535–552, Springer, 2007.

[20] BERTHOME, P., FECHEROLLE, T., GUILLOTEAU, N., and LALANDE, J.-F., "Repackaging android applications for auditing access to private data," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pp. 388–396, IEEE, 2012.

[21] BIBA, K. J., "Integrity considerations for secure computer systems," tech. rep., DTIC Document, 1977.

[22] BÖHMER, M., HECHT, B., SCHÖNING, J., KRÜGER, A., and BAUER, G., "Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage," in *Proceedings of the 13th international conference on Human computer interaction with mobile devices and services*, pp. 47–56, ACM, 2011.

[23] BONEH, D., CRESCENZO, G. D., OSTROVSKY, R., and PERSIANO, G., "Public key encryption with keyword search," in *EUROCRYPT* (CACHIN, C. and CAMENISCH, J., eds.), vol. 3027 of *Lecture Notes in Computer Science*, pp. 506–522, Springer, 2004.

[24] BORDERS, K., VANDER WEELE, E., LAU, B., and PRAKASH, A., "Protecting confidential data on personal computers with storage capsules," *Ann Arbor*, vol. 1001, p. 48109, 2009.

[25] BORISOV, N., GOLDBERG, I., and BREWER, E., "Off-the-record communication, or, why not to use pgp," in *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pp. 77–84, ACM, 2004.

[26] BRICKELL, E. and LI, J., "Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities," in *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pp. 21–30, 2007.

[27] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., and SHASTRY, B., "Towards taming privilege-escalation attacks on android," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2012.

[28] CBS INTERACTIVE INC., "Snowden: Leak of NSA spy programs "marks my end"." http://www.cbsnews.com/8301-201_162-57588462/snowden-leak-of-nsa-spy-programs-marks-my-end/.

[29] CHANG, Y.-C. and MITZENMACHER, M., "Privacy preserving keyword searches on remote encrypted data," in *Applied Cryptography and Network Security* (IOANNIDIS, J., KEROMYTIS, A., and YUNG, M., eds.), vol. 3531 of *Lecture Notes in Computer Science*, pp. 442–455, Springer, 2005.

[30] CHEN, Y., REYMONDJOHNSON, S., SUN, Z., and LU, L., "Shreds: Fine-grained execution units with private memory," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, (San Jose, CA), pp. 56–71, May 2016.

[31] CHIN, E., FELT, A. P., GREENWOOD, K., and WAGNER, D., "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 239–252, ACM, 2011.

[32] CHROME: DEVELOPER, "Google Chrome Mobile FAQ." https://developers.google.com/chrome/mobile/docs/faq.

[33] CHUNG, K.-M., KALAI, Y., and VADHAN, S., "Improved delegation of computation using fully homomorphic encryption," in *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, (Berlin, Heidelberg), pp. 483–501, Springer-Verlag, 2010.

[34] COHEN, W. W., "Enron email dataset." http://www.cs.cmu.edu/ enron, August 2009.

[35] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., and WARFIELD, A., "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, (Cascais, Portugal), Oct. 2011.

[36] COOK, K., "[patch] security: Yama lsm." http://lwn.net/Articles/393012/.

[37] COSTAN, V. and DEVADAS, S., "Intel sgx explained," tech. rep., Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org.

[38] CUI, W., KATZ, R. H., and TIAN TAN, W., "Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, (Tucson, AZ), Dec. 2005.

[39] CURTMOLA, R., GARAY, J. A., KAMARA, S., and OSTROVSKY, R., "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Oct.–Nov. 2006.

[40] D. J. WALKER-MORGAN, "Sniffer tool displays other people's WhatsApp messages." http://www.h-online.com/security/news/item/Sniffer-tool-displays-other-people-s-WhatsApp-messages-1574382.html.

[41] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., and WINANDY, M., "Privilege escalation attacks on android," in *Information Security*, pp. 346–360, Springer, 2011.

[42] DAVIDSON, L., "Windows 7 uac whitelist: Proof-of-concept source code." http://www.pretentiousname.com/misc/W7E_Source/win7_uac_poc_details.html.

[43] DELTCHEVA, R., "Apple, AT&T data leak protection issues latest in cloud failures." http://www.messagingarchitects.com/resources/security-compliance-news/email-security/apple-att-data-leak-protection-issues-latest-in-cloud-failures19836720.html, June 2010.

[44] DINGLEDINE, R., MATHEWSON, N., and SYVERSON, P., "Tor: The second-generation onion router," tech. rep., DTIC Document, 2004.

[45] DONG, X., CHEN, Z., SIADATI, H., TOPLE, S., SAXENA, P., and LIANG, Z., "Protecting sensitive web content from client-side vulnerabilities with cryptons," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, (Berlin, Germany), Oct. 2013.

[46] EDWARDS, J., "There's a huge password security quirk in ios 7 that lets siri control your iphone." http://www.businessinsider.com/password-security-flaw-in-ios-7-lets-siri-control-your-iphone-2013-9.

[47] ELKINS, M., "Mime security with pretty good privacy (pgp)," 1996.

[48] ELSON, J. and CERPA, A., "RFC 3507 - Internet Content Adaptation Protocol (ICAP)." http://www.ietf.org/rfc/rfc3507.txt.

[49] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., and SHETH, A., "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones.," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Vancouver, Canada), Oct. 2010.

[50] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., and SMITH, M., "Why eve and mallory love android: An analysis of android ssl (in)security," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, (Raleigh, NC), Oct. 2012.

[51] FAHL, S., HARBACH, M., MUDERS, T., and SMITH, M., "Trustsplit: usable confidentiality for social network messaging," in *Proceedings of the 23rd ACM conference on Hypertext and social media*, pp. 145–154, ACM, 2012.

[52] FARB, M., LIN, Y.-H., KIM, T. H.-J., MCCUNE, J., and PERRIG, A., "Safeslinger: easy-to-use and secure public-key exchange," in *Proceedings of the 19th annual international conference on Mobile computing & networking*, pp. 417–428, ACM, 2013.

[53] FARQUHAR, I., "Engineering Security Solutions at Layer 8 and Above." https://blogs.rsa.com/engineering-security-solutions-at-layer-8-and-above/, December 2010.

[54] FAULKNER, L., "Beyond the five-user assumption: Benefits of increased sample sizes in usability testing," *Behavior Research Methods, Instruments, & Computers*, vol. 35, no. 3, pp. 379–383, 2003.

[55] FELDMAN, A. J., BLANKSTEIN, A., FREEDMAN, M. J., and FELTEN, E. W., "Social networking with frientegrity: privacy and integrity with an untrusted provider," in *Proceedings of the 21st USENIX Security Symposium (Security)*, (Bellevue, WA), Aug. 2012.

[56] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., and CHIN, E., "Permission re-delegation: Attacks and defenses.," in *Proceedings of the 20th USENIX Security Symposium (Security)*, (San Francisco, CA), Aug. 2011.

[57] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., and BONEH, D., "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, (Bolton Landing, NY), Oct. 2003.

[58] GARFINKEL, T. and ROSENBLUM, M., "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2003.

[59] GENNARO, R., GENTRY, C., and PARNO, B., "Non-interactive verifiable computing: outsourcing computation to untrusted workers," in *Proceedings of the 30th annual*

conference on Advances in cryptology, CRYPTO'10, (Berlin, Heidelberg), pp. 465–482, Springer-Verlag, 2010.

[60] GENTRY, C., *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[61] GIBBERBOT, "Gibberbot for Android devices." https://securityinabox.org/en/Gibberbot_main.

[62] GNOME DEV CENTER, "ATK - Accessibility Toolkit." https://developer.gnome.org/atk/2.8/.

[63] GO LAUNCHER DEV TEAM, "Go launcher ex notification." https://play.google.com/store/apps/details?id=com.gau.golauncherex.notification.

[64] GOH, E.-J., "Secure indexes," *IACR Cryptology ePrint Archive*, 2003.

[65] GOLDREICH, O. and OSTROVSKY, R., "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[66] GOLDWASSER, S., MICALI, S., and RACKOFF, C., "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, pp. 186–208, Feb. 1989.

[67] GOLDWASSER, S., KALAI, Y. T., and ROTHBLUM, G. N., "Delegating computation: interactive proofs for muggles," in *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, (New York, NY, USA), pp. 113–122, ACM, 2008.

[68] GOOGLE, INC., "Google Accessibility." https://www.google.com/accessibility/policy/.

[69] GOOGLE INC., "Section 508 Compliance (VPAT)." https://www.google.com/sites/accessibility.html.

[70] GRACE, M., ZHOU, Y., WANG, Z., and JIANG, X., "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2012.

[71] GUHA, S., TANG, K., and FRANCIS, P., "Noyb: Privacy in online social networks," in *Proceedings of the first workshop on Online social networks*, pp. 49–54, ACM, 2008.

[72] GUMMADI, R., BALAKRISHNAN, H., MANIATIS, P., and RATNASAMY, S., "Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (Boston, MA), Apr. 2009.

[73] HAN, J., OWUSU, E., NGUYEN, L., PERRIG, A., and ZHANG, J., "Accomplice: Location inference using accelerometers on smartphones," in *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pp. 1–9, Jan 2012.

[74] HENRY, S., "Largest hacking, data breach prosecution in U.S. history launches with five arrests." http://www.mercurynews.com/business/ci23730361/largest-hacking-data-breach-prosecution-u-s-history, July 2013.

[75] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., and DEL CUVILLO, J., "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, (Tel-Aviv, Israel), pp. 1–8, 2013.

[76] HOHMUTH, M., PETER, M., HARTIG, H., and SHAPIRO, J. S., "Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors," in *Proc. of the ACM SIGOPS European Workshop*, 2004.

[77] HUNT, T., ZHU, Z., XU, Y., PETER, S., and WITCHEL, E., "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA), Nov. 2016.

[78] IMSPECTOR, "IMSpector: Instant Messenger Proxy Service." `http://www.imspector.org/wordpress/`.

[79] INTEL, "Graphics Drivers Blue-ray Disc* Playback On Intel Graphics FAQ." `http://www.intel.com/support/graphics/sb/CS-029871.htm#bestexperience`, 2008. Accessed: 05/04/2015.

[80] INTEL CORPORATION, "Intel Software Guard Extensions Programming Reference (rev1)," Sept. 2013. 329298-001US.

[81] INTEL CORPORATION, "Intel Software Guard Extensions Programming Reference (rev2)," Oct. 2014. 329298-002US.

[82] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, "International technology - Open Systems Interconnection - Basic Reference Model: The Basic Model." http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf.

[83] IT BUSINESS EDGE, "Ten Mistakes That Can Ruin Customers' Mobile App Experience." http://www.itbusinessedge.com/slideshows/show.aspx?c=96038.

[84] JANA, S. and SHMATIKOV, V., "Memento: Learning secrets from process footprints," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), May 2012.

[85] JANG, Y., CHUNG, S. P., PAYNE, B. D., and LEE, W., "Gyrus: A Framework for User-Intent Monitoring of Text-based Networked Applications," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2014.

[86] JANG, Y., SONG, C., CHUNG, S. P., WANG, T., and LEE, W., "A11y Attacks: Exploiting Accessibility in Operating Systems," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, (Scottsdale, Arizona), Nov. 2014.

[87] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., and MILLSTEIN, T., "Dr. android and mr. hide: fine-grained permissions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pp. 3–14, ACM, 2012.

[88] JIANG, X., "Gingermaster: First android malware utilizing a root exploit on android 2.3 (gingerbread)." `http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/`.

[89] JIANG, X., WANG, X., and XU, D., "Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Oct.–Nov. 2007.

[90] JOHNSON, S., SCARLATA, V., ROZAS, C., BRICKELL, E., and MCKEEN, F., "Intel software guard extensions: Epid provisioning and attestation services," *White Paper*, 2016.

[91] JOSHI, A., KING, S. T., DUNLAP, G. W., and CHEN, P. M., "Detecting past and present intrusions through vulnerability-specific predicates," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, (Brighton, UK), Oct. 2005.

[92] KACHOLD, L., "Layer 8 Linux Security." http://www.linuxgazette.net/166/kachold.html, July 2009.

[93] KAMARA, S., PAPAMANTHOU, C., and ROEDER, T., "Dynamic searchable symmetric encryption," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, (Raleigh, NC), Oct. 2012.

[94] KIM, S., SHIN, Y., HA, J., KIM, T., and HAN, D., "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, (Philadelphia, PA), Nov. 2015.

[95] KING, S. T., TUCEK, J., COZZIE, A., GRIER, C., JIANG, W., and ZHOU, Y., "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, (Berkeley, CA, USA), pp. 5:1–5:8, USENIX Association, 2008.

[96] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., and WINWOOD, S., "sel4: formal verification of an os kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, (Big Sky, MT), Oct. 2009.

[97] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., and WINWOOD, S., "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, 2009.

[98] KOBEISSI, N., "Cryptocat." https://crypto.cat.

[99] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., and VARADHARAJAN, V., "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, p. 10, ACM, 2014.

[100] KONTAXIS, G., POLYCHRONAKIS, M., KEROMYTIS, A. D., and MARKATOS, E. P., "Privacy-preserving Social Plugins," in *Proceedings of the 21st USENIX Security Symposium (Security)*, (Bellevue, WA), Aug. 2012.

[101] LAU, B., CHUNG, P. H., SONG, C., JANG, Y., LEE, W., and BOLDYREVA, A., "Mimesis Aegis: A Mimicry Privacy Shield," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, (San Diego, CA), Aug. 2014.

[102] LAU, B., CHUNG, S., SONG, C., JANG, Y., LEE, W., and BOLDYREVA, A., "Mimesis Aegis: A Mimicry Privacy Shield." http://hdl.handle.net/1853/52026.

[103] LAU, B., JANG, Y., SONG, C., WANG, T., CHUNG, P. H., and ROYAL, P., "Mactans: Injecting malware into iOS devices via malicious chargers," in *Black Hat USA Briefings (Black Hat USA)*, (Las Vegas, NV), Aug. 2013.

[104] LI, W., MA, M., HAN, J., XIA, Y., ZANG, B., CHU, C.-K., and LI, T., "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, p. 8, ACM, 2014.

[105] LU, L., LI, Z., WU, Z., LEE, W., and JIANG, G., "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, (Raleigh, NC), Oct. 2012.

[106] LUCAS, M. M. and BORISOV, N., "Flybynight: mitigating the privacy risks of social networking," in *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, pp. 1–8, ACM, 2008.

[107] MAC OSX DEVELOPER CENTER, "NSAccessibility Protocol Reference." `https://developer.apple.com/library/mac/#documentation/Cocoa/`

Reference/ApplicationKit/Protocols/NSAccessibility_Protocol/Reference/Reference.html.

[108] MADNICK, S. E. and DONOVAN, J. J., "Application and Analysis of The Virtual Machine Approach to Information System Security and Isolation," in *Proc of the Workshop on Virtual Computer Systems*, 1973.

[109] MARTIGNONI, L., POOSANKAM, P., ZAHARIA, M., HAN, J., MCCAMANT, S., SONG, D., PAXSON, V., PERRIG, A., SHENKER, S., and STOICA, I., "Cloud terminal: secure access to sensitive applications from untrusted systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, (Boston, MA), June 2012.

[110] MARTIGNONI, L., POOSANKAM, P., ZAHARIA, M., HAN, J., MCCAMANT, S., SONG, D., PAXSON, V., PERRIG, A., SHENKER, S., and STOICA, I., "Cloud Terminal: Secure access to sensitive applications from untrusted systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, (Boston, MA), pp. 165–182, June 2012.

[111] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., and PERRIG, A., "Trustvisor: Efficient tcb reduction and attestation," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2010.

[112] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., and PERRIG, A., "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), pp. 143–158, May 2010.

[113] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., and ISOZAKI, H., "Flicker: An Execution Infrastructure for TCB Minimization," in *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, (Glasgow, Scotland), pp. 315–328, Mar. 2008.

[114] MCLAWHORN, C., "Recent development: Leveling the accessibility playing field: Section 508 of the rehabilitation act," *NORTH CAROLINA JOURNAL OF LAW & TECHNOLOGY*, vol. 3, no. 1, pp. 63–100, 2001.

[115] MICROSOFT, "Windows integrity mechanism design." http://msdn.microsoft.com/en-us/library/bb625963.aspx.

[116] MICROSOFT, "Windows vista integrity mechanism technical reference." http://msdn.microsoft.com/en-us/library/bb625964.aspx.

[117] MICROSOFT CORPORATION, "Microsoft and section 508." http://www.microsoft.com/enable/microsoft/section508.aspx.

[118] MICROSOFT CORPORATION, "User account control." http://windows.microsoft.com/en-us/windows7/products/features/user-account-control.

[119] MICROSOFT DEVELOPER NETWORK, "Inspect." http://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx.

[120] MICROSOFT DEVELOPER NETWORK, "UI Automation Overview." http://msdn.microsoft.com/en-us/library/ms747327.aspx.

[121] MIT, "MIT PGP Public Key Server." http://pgp.mit.edu/.

[122] MOTIEE, S., HAWKEY, K., and BEZNOSOV, K., "Do windows users follow the principle of least privilege?: investigating user account control practices," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, (New York, NY, USA), ACM, 2010.

[123] MOTOROLA INC., "Moto X Features: Touchless Control." http://www.motorola.com/us/Moto-X-Features-Touchless-Control/motox-features-2-touchless.html.

[124] MRTON, N., "USBIP protocol documentation," June 2011. https://lwn.net/Articles/449509/.

[125] ONARLIOGLU, K., MULLINER, C., ROBERTSON, W., and KIRDA, E., "Privexec: Private execution as an operating system service," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), pp. 206–220, May 2013.

[126] OPEN WHISPER SYSTEMS, "Secure texts for Android." https://whispersystems.org.

[127] OU, G., "Vista Speech Command exposes remote exploit." http://www.zdnet.com/blog/ou/vista-speech-command-exposes-remote-exploit/416.

[128] PARNO, B., HOWELL, J., GENTRY, C., and RAYKOVA, M., "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), May 2013.

[129] PEEK, D. and FLINN, J., "Trapperkeeper: the case for using virtualization to add type awareness to file systems," in *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, pp. 8–8, USENIX Association, 2010.

[130] PEW INTERNET, "What Internet Users Do On A Typical Day." http://www.pewinternet.org/Static-Pages/Trend-Data-(Adults)/Online-Activities-Daily.aspx.

[131] PEW INTERNET, "What Internet Users Do Online." http://www.pewinternet.org/Static-Pages/Trend-Data-(Adults)/Online-Activites-Total.aspx.

[132] POPS, "Pops ringtons & notifications." https://play.google.com/store/apps/details?id=com.pops.app.

[133] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., and COWAN, C., "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), May 2012.

[134] ROZAS, C., "Intel software guard extensions," Nov. 2013. http://www.pdl.cmu.edu/SDI/2013/slides/rozas-SGX.pdf.

[135] SCHLEGEL, R., ZHANG, K., YONG ZHOU, X., INTWALA, M., KAPADIA, A., and WANG, X., "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2011.

[136] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., and RUSSINOVICH, M., "VC3: Trustworthy Data Analytics in the Cloud using SGX," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, (San Jose, CA), May 2015.

[137] SESHADRI, A., LUK, M., QU, N., and PERRIG, A., "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[138] SETTY, S., MCPHERSON, R., BLUMBERG, A. J., and WALFISH, M., "Making argument systems for outsourced computation practical (sometimes)," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2012.

[139] SETTY, S., VU, V., PANPALIA, N., BRAUN, B., BLUMBERG, A. J., and WALFISH, M., "Taking proof-based verified computation a few steps closer to practicality," in *Proceedings of the 21st USENIX Security Symposium (Security)*, (Bellevue, WA), Aug. 2012.

[140] SHACKLEFORD, D., "Blind as a Bat? Supporting Packet Decryption for Security Scanning." http://www.sans.org/reading_room/analysts_program/vss-BlindasaBat.pdf.

[141] SHAH, K., "Common Mobile App Design Mistakes to Take Care." http://www.enterprisecioforum.com/en/blogs/kaushalshah/common-mobile-app-design-mistakes-take-c.

[142] SHENG, S., BRODERICK, L., HYLAND, J., and KORANDA, C., "Why johnny still can't encrypt: evaluating the usability of email encryption software," in *Symposium On Usable Privacy and Security*, 2006.

[143] SHIH, M.-W., KUMAR, M., KIM, T., and GAVRILOVSKA, A., "S-nfv: Securing nfv states by using sgx," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 45–48, ACM, 2016.

[144] SHINAGAWA, T., EIRAKU, H., TANIMOTO, K., OMOTE, K., HASEGAWA, S., HORIE, T., HIRANO, M., KOURAI, K., OYAMA, Y., KAWAI, E., KONO, K., CHIBA, S., SHINJO, Y., and KATO, K., "Bitvisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[145] SHNEIDERMAN, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, fourth ed., 2005.

[146] SONG, D. X., WAGNER, D., and PERRIG, A., "Practical techniques for searches on encrypted data," in *Proceedings of the 21st IEEE Symposium on Security and Privacy (Oakland)*, (Oakland, CA), May 2000.

[147] STEVE SLAVEN, "xautomation." `http://hoopajoo.net/projects/xautomation.html`.

[148] SYMANTAC CORPORATION, "Symantec desktop email encryption end-to-end email encryption software for laptops and desktops." http://www.symantec.com/desktop-email-encryption.

[149] THE CHROMIUM PROJECTS, "Benchmarking Extension." `http://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/chrome-benchmarking-extension`.

[150] THE UNITED STATES GOVERNMENT, "Section 508 Of The Rehabilitation Act." `http://www.section508.gov/Section-508-Of-The-Rehabilitation-Act`.

[151] THOMAS BARNETT, JR., "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018." http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[152] TSAI, C.-C., PORTER, D. E., and VIJ, M., "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[153] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., and MIYAUCHI, H., "Cryptanalysis of des implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 62–76, Springer, 2003.

[154] US-CERT/NIST, "Cve-2013-4787." `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787`.

[155] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., and DATTA, A., "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), pp. 430–444, May 2013.

[156] WALTER, S., "Proxsmtp: An smtp filter." http://memberwebs.com/stef/software/proxsmtp/.

[157] WANG, T., LU, K., LU, L., CHUNG, S., and LEE, W., "Jekyll on ios: When benign apps become evil," in *Proceedings of the 22th USENIX Security Symposium (Security)*, (Washington, DC), Aug. 2013.

[158] WESSELS, D., NORDSTRÖM, H., ROUSSKOV, A., CHADD, A., COLLINS, R., SERASSIO, G., WILTON, S., and FRANCESCO, C., "Squid: Optimising web delivery." http://www.squid-cache.org/.

[159] WHITTAKER, S., MATTHEWS, T., CERRUTI, J., BADENES, H., and TANG, J., "Am I Wasting My Time Organizing Email?: a Study of Email Refinding," in *Proceedings of the 2011 annual conference on Human factors in computing systems*, pp. 3449–3458, ACM, 2011.

[160] WHITTEN, A. and TYGAR, J. D., "Why Johnny cant encrypt: A usability evaluation of PGP 5.0," in *Proceedings of the 8th USENIX Security Symposium (Security)*, (Washington, DC), Aug. 1999.

[161] WOJTCZUK, R. and TERESHKIN, A., "Attacking intel® bios," *Invisible Things Lab*, 2010.

[162] WU, C., WANG, Z., and JIANG, X., "Taming Hosted Hypervisors with (Mostly) Deprivileged Execution," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2013.

[163] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., and JIANG, X., "Airbag: Boosting smartphone resistance to malware infection," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2014.

[164] WU, L., GRACE, M., ZHOU, Y., WU, C., and JIANG, X., "The impact of vendor customizations on Android security," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, (Berlin, Germany), Oct. 2013.

[165] XU, N., ZHANG, F., LUO, Y., JIA, W., XUAN, D., and TENG, J., "Stealthy video capturer: A new video-based spyware in 3g smartphones," in *Proceedings of the Second ACM Conference on Wireless Network Security*, WiSec '09, (New York, NY, USA), ACM, 2009.

[166] XU, R., SAÏDI, H., and ANDERSON, R., "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Security Symposium (Security)*, (Bellevue, WA), Aug. 2012.

[167] ZHOU, Y. and JIANG, X., "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2013.

[168] ZHOU, Z., GLIGOR, V. D., NEWSOME, J., and MCCUNE, J. M., "Building verifiable trusted path on commodity x86 computers," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 616–630, IEEE, 2012.

[169] ZHOU, Z., YU, M., and GLIGOR, V. D., "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 308–323, IEEE, 2014.