Gyrus: A Framework for User-Intent Monitoring of Text-Based Networked Applications

Yeongjin Jang, Simon P. Chung
Georgia Institute of Technology
yeongjin.jang@gatech.edu, pchung34@mail.gatech.edu

Bryan D. Payne
Nebula, Inc.
bdpayne@acm.org

Wenke Lee
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract—**Traditional security systems have largely focused on attack detection. Unfortunately, accurately identifying the latest attack has proven to be a never-ending cycle. In this paper, we propose a way to break this cycle by ensuring that a system's behavior matches the user's intent. Since our approach is attack agnostic, it will scale better than traditional security systems.**

There are two key components to our approach. First, we capture the user's *intent* through their interactions with an application. Second, we verify that the resulting system *output* can be mapped back to the user's interactions. To demonstrate how this works we created *Gyrus*, a research prototype that observes user interactions for common tasks such as sending email, instant messaging, online social networking, and online financial services. Gyrus secures these applications from malicious behavior such as spam and wire fraud by allowing only outgoing traffic with content that matches the user's intent. To understand how Gyrus captures user intent, consider the case of a text-based application. In this case the user's input is displayed on the screen so the user can confirm that their input is correct. Gyrus builds on this concept by focusing on what is being displayed to the user instead of what the user has typed or clicked. We call this the "what you see is what you send (WYSIWYS)" policy. We implemented Gyrus under a standard virtualization environment, and our prototype system successfully stops malware from sending unintended content over the network. Our evaluation shows that Gyrus is very efficient and introduces no noticeable delay to a users' interaction with the protected applications.

I. Introduction

Host-based security systems have traditionally focused on detecting attacks. Misuse detection targets attacks that follow a predefined malicious pattern, whereas anomaly detection identifies attacks as anything that *cannot* be the result of correct execution under any input or execution environment. It has been shown over time that systems following this approach usually have too narrow a definition of "attacks"[1]; misuse detection generally cannot detect new attacks, while anomaly detection are known to suffer from mimicry attacks.

[1]Usually necessary to keep false positive rate acceptable.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/doi-info-to-be-provided-later

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 behavior 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 [30]), the click captures all the semantics of the user's intent (e.g. access the camera), in other cases (e.g. BINDER [6], Not-a-Bot [15]), the user's intent involves a richer context, and failure to capture the full semantics will again allow for attacks to disguise as 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 work, we propose a way to capture 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*[2] 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[3]). 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 dom-U display window (the VNC viewer in KVM environment), covering editable text area of targeted

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

[3]In this paper, 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.

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 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 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, Gyrus will make sure the outgoing network traffic matches the text displayed. In short, Gyrus enforces 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. Also, 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 for assistive technology called UI Automation. As for the logic to map user intent to 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 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. Section V discuses what applications are best protected by Gyrus.

The per-application development cost for Gyrus is justifiable since Gyrus is attack-agnostic: it 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 our work include 1) the "What You See Is What You Send" concept which 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[4] 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. Our prototype currently supports email, instant messaging, social networking applications, and online financial applications, effectively covering the most common network applications in everyday use.

The rest of this paper is organized as follows: Section II discusses related work. Section III discusses the Gyrus threat model and the rationale of our What You See Is What You Send policy. Section IV presents the architecture and implementation details for Gyrus. Section V demonstrates how Gyrus can be used for real world applications. Section VI presents the evaluation of the Gyrus framework, and finally we conclude in Section VII.

II. 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.

A. Capturing Human Intent

Like Gyrus, BINDER [6] and Not-A-Bot (NAB) [15] 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 which come shortly after the user input is user intended.* However, as stated in the introduction, 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. Also, 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

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

events, and by decoupling the capturing of user intent from the enforcement of its traffic filtering policy.

User-Driven Access Control [30] 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 a 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.

B. *Trusted Execution Environment*

Virtualization has enjoyed 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 [22], [16], and recent research has aimed to enhance this security by reducing the size of the hypervisor's code [34], [40], modularizing its components [5], or verifying its security [20]. These isolation properties make virtualized environments an attractive way to implement security applications. Virtualization-based solutions have been used to implement trusted computing architectures [8], [24], intrusion detection systems [9], malware analysis systems [18], and zero-day intrusion analysis systems [19]. However, none of these take user intent into account and 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.

C. *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 [13], [14], [10], Probabilistically Checkable Proofs (PCPs) [3], [31], [32], or relying on fully-homomorphic encryption (FHE) [11], [4]. 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 a 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 [27] 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.

III. 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.

A. *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 monitor. We make the following security assumptions:

1) The hypervisor and dom-0 are fully trusted.
2) Attackers cannot have physical access to the machine, and we trust the hardware.
3) 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.
4) 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 a *Secure Overlay* to verify the validity of this information. Detailed information for these components will be described in the next section.

B. *User Intent*

As mentioned in the introduction, the goal of Gyrus is to capture rich semantics to understand a user's intent. This 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. In order to make a sound security decision, we must capture more detailed information 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.

C. *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,

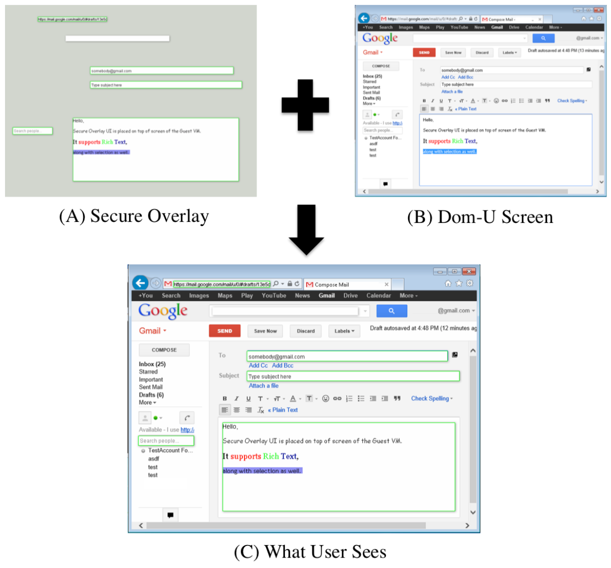
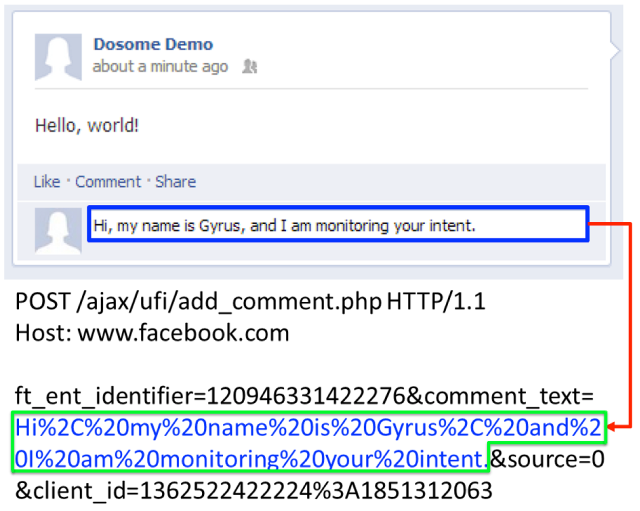


Fig. 1. Secure Overlay working with the GMail application in Internet Explorer 10. Overlaid edit controls are highlighted with green bounding box. Gyrus changes the border color to red if it detects any infringement.



Dosome Demo
about a minute ago

Hello, world!

Like · Comment · Share

Hi, my name is Gyrus, and I am monitoring your intent.

```
POST /ajax/ufi/add_comment.php HTTP/1.1
Host: www.facebook.com

ft_ent_identifier=120946331422276&comment_text=
Hi%2C%20my%20name%20is%20Gyrus%2C%20and%2
0I%20am%20monitoring%20your%20intent.&source=0
&client_id=1362522422224%3A1851312063
```

Fig. 2. An example WYSIWYS applicable operation: Facebook comment. After adding a comment and pressing the ENTER key, the application generates network traffic going to www.facebook.com/ajax/ufi/add-comment.php. There is a direct mapping of user-intended content between on-screen text and outgoing traffic.

the text that the user types will be displayed on the screen. This 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. In addition, Gyrus needs information about the UI structure. In Gyrus, we have two components that implement these features: a dom-U component called *UI Monitor* extracts textual content and a high-level UI structure of the current screen, and a dom-0 trusted component called *Secure Overlay* to verify if the captured text matches the user's intent.

The *UI Monitor* operates on top of the *UI Automation* [26] 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 screen, and fits our purpose very well). Not only does the UI monitor capture the displayed text, 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 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 exactly the 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. In addition, it can support text-editing features such as cut/copy/paste, automatic spell correction, selection of text from combo box, etc.

D. 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 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 largely 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 certain URLs that submit user-intended contents, such as posting Twitter messages, adding comments on Facebook, or sending money on Paypal. In Section IV, we will describe how to identify such traffic using the *User Intent Signature*.

E. 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 be generated to download all the images on the loaded pages. Moreover, 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 paper, 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 about 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 for the user (e.g., financial loss). Examples of such transactions include:

1) Transferring money through an online financial service.
2) Modifying value fields (e.g., speed of a turbine, or level of the water in a nuclear power plant) of SCADA (Supervisory Control And Data Acquisition) systems.
3) Sending a message through an e-mail client, or an internet messaging (IM) application.
4) 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 Section V. 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, it 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.

IV. Design and Implementation

A. Architecture

Gyrus employs a virtual machine based isolation mechanism; therefore, its architecture is separated in 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. It 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 *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 in 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. It inspects all traffic under the monitored protocol. If outgoing traffic is using a protocol corresponding to any of the applications protected by Gyrus, traffic is inspected by querying the *Authorization DB* to see if the traffic is intended by the user. Unintended traffic is blocked. Also note that the *Network Monitor* will allow all traffic from other protocols 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 we designed specifically for Gyrus. It covers three categories of information: the condition that triggers network traffic, the required UI structure data for catching content-intent, and the content of monitored traffic. This user-intent signature language represents our effort to simplify and provide structures to the development of per-application logic under Gyrus.

Central Control *Central Control* contains the logic that runs the other elements. Its main task is to process intercepted

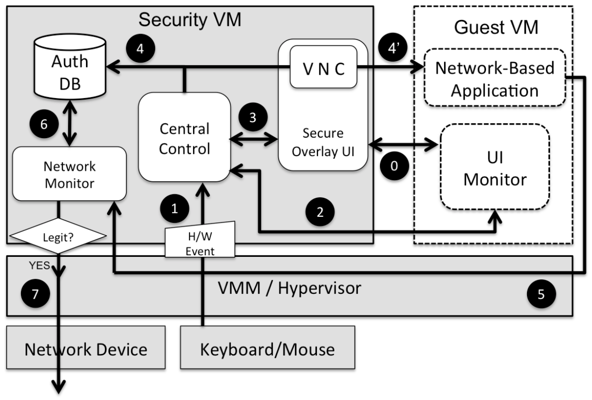


Fig. 3. Workflow of Gyrus upon receiving a traffic-triggering event. Grayed and solid-lined areas are trusted components, while dotted lines indicate untrusted components.

hardware input events. Upon arrival of these events, the *Central Control* will query the UI monitor to see if the event signifies user intent to send the currently displayed content out to the network. If so, the *Central Control* will query the *Secure Overlay* and *User-Intent Signature* to generate an authorization 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 alter any inputs, it does not alter the user experience beyond adding an imperceptible delay (see Table II).

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). Central Control queries the UI Monitor to see if this input triggers network traffic or not (2). If it does, Central Control queries the Secure Overlay (3) to create a dynamic authorization vector that describes the user-intended outgoing traffic and save it to 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). Traffic is intercepted and inspected by the network monitor. The network monitor queries the Authorization DB to determine if the intercepted traffic matches user intent (6). Traffic will be allowed if it matches an authorization vector. Otherwise, traffic is blocked and Gyrus raises an alarm to notify the user of a likely attack attempt (7).

B. Implementation

We implemented our prototype of the Gyrus framework using a Linux / KVM host running Ubuntu 12.04.2 LTS and a dom-U running Windows 7 SP1. We note that the Gyrus architecture is not limited to this specific 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 [34] as a lightweight secure hypervisor, or could use the Dalvik VM [2] on Android as an isolation and hardware event-capturing instrument. Similarly, the *UI Monitor* is not limited to the UI Automation on Microsoft platforms. Other accessibility frameworks – such as ATK [12] and XAutomation [36] on Linux or NSAccessiblity [21] on Mac OS X – could replace it. Finally, Gyrus could be implemented using a thin-client model with the trusted client terminal [23] and a network monitor on the remote host.

1) In-Guest UI Monitor:

Since our implementation of the UI monitor is largely 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 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 certain 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 Section III, the UI monitor is a component that runs in dom-U, and it serves two purposes: to determine if a keyboard/mouse input event[5] signifies the user’s intent to send something over the network, and to provide information to the 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 whether current input generates network traffic, the UI Monitor first looks for the UI object that receives the current input. To determine if a mouse click, for example, signifies a user’s intent to generate outgoing traffic, the UI Monitor calls a function named `ElementFromPoint` to get the object that is currently located under the cursor. For the keystroke events, we use the `GetFocusedElement` function 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 text box of interest by querying the UIAutomation library for the properties of this object. Application-specific logic required for determining the traffic-triggering event is configured with a *User Intent Signature* (e.g. checking whether it is a button with its name being `Send` on 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*

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

Overlay that the traffic-triggering event has occurred. The *Secure Overlay* also receives details about what operation and which application triggered the event, and the content needed 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. Also, 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. Section VI-A 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. At first, before extracting the currently displayed text, the UI Monitor registers the text box to be monitored with the *Secure Overlay* for tracking 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 form UI object 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, the corresponding information is extracted from the `TextPattern` object. Finally, the UI Monitor catches 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 subscribe to the `PropertyChangedEvent` of target object for the `Value` property of the object, and in the event handler, we send the updated content to the overlay. This 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 we can display the caret properly.

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 on the overlay the text content of the currently focused window; this policy significantly simplifies our implementation and only has a small impact on the usability of our system[6]. Although overlaid text boxes for background applications are not displayed, the *Secure Overlay* maintains previously captured user-intended text while it is visible, and disables its update while it is hidden[7]. Therefore, the text integrity of background applications can be preserved 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 content of the window that is closed or has lost focus, and 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 target applications when the text boxes of interest in these applications first receive focus.

2) 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 dom-U *UI monitor*. This part mainly involves some UI/graphics programming, and some book keeping 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 are rendered without noticeable 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, so that we can determine what the user intended outgoing traffic should look like when the user finally 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 *Central Control*. Based on 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 screen, this is very easily achieved. However, the task is more complicated if the text is displayed in a text box with scrollbar. In this case, the UI monitor is still able to capture all the text in the textbox; 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

[6]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.

[7]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.

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:

1) 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/deleting a single character/chunk of text).
2) Updates can only occur in the visible part of text (i.e., the point where the character or chunk of text is inserted or deleted must be visible before the update occurs).
3) If a chunk of text is inserted, the end of the chunk must be visible after the update. Similarly, if one character is inserted, the character must be visible after the update.
4) 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 problem. In this case, the user could check the text displayed by the overlay to determine if her intent was properly captured by Gyrus. If it was, she can commit the input to the network. The above design allows us to correctly and securely handle most normal 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).*

3) Authorization DB: The Authorization DB saves the user intent captured by the secure overlay at the time 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 actual outgoing traffic of the corresponding protocol is observed. To allow for efficient lookup by the network monitor, 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 intend the same message to be sent multiple times.

4) 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* [38] and *IMSpector* [17], respectively.

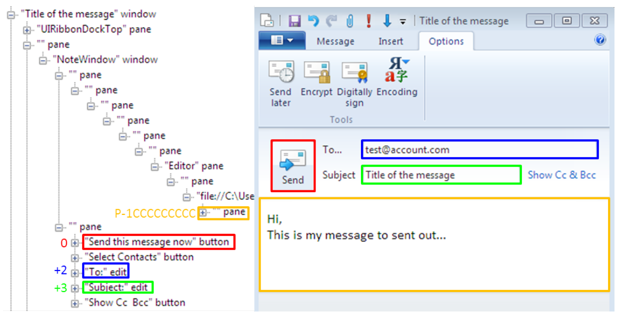


Fig. 4. UI structure of Windows Live Mail. Tree structure on the left is from Inspect.exe. '0' indicates event-receiving object (send button), +2 and +3 indicate 2nd and 3rd sibling from the origin (negative number indicates previous sibling). 'P' is a symbol for a parent, and 'C' refers to child.

For HTTP, even though there exists a transparent proxy with the capability of ICAP [7] handling such as *Squid* [39], we wrote our own implementation due to performance issue[8]. 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 afterwards. 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 with IDSs/IPSs [33].

5) User-Intent Signature: As we have mentioned in the introduction, 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 Example 1, and Example 2 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 *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 `ENTER` for reacting on pressing return key on the message dialog of an instant messenger application. The traffic-triggering event is only partially defined by this field. It should be linked with application-specific, operation-

[8]Squid did not support multi-threading for traffic relaying. It can cause severe delays when a web browser loads a web page.

Example 1 User Intent Signature for sending e-mail on Windows Live Mail.

```
{
  "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"
      }
    }
}
```

Example 2 User Intent Signature for posting comments on Facebook Web-app.

```
{
  "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"
    }
  }
}
```

specific information to correctly identify whether the input will be delivering to the specified application. For correctly figuring out the details from an input receiving application, we use the tree-structure information of UI, 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. Example 1 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 this tree-structure is formed in UIAutomation. Note that under the `COND` region, all of the conditions are listed. For internal fields, the number indicates the relative distance from an event receiving object as siblings on the tree. So `'0'` means the object specified triggered the event, and `'+2'` or `'+3'` indicates the next siblings at the specified distance (negative number indicates previous sibling). `'P'` and `'C'` refers to parent and 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. We use the `TYPE` field to specify 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[9], and both `METHOD` and `URL`[10] fields are used for web-apps (`METHOD` is for distinguishing 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 *Central Control*, then *Central Control* extracts verified text from the *Secure Overlay*, and finally the authorization vector will be created and saved based on this information. 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

[9]We assigned natural names for each operation. METHOD = SEND in Example 1 means that the signature should only monitor sending operations in the SMTP protocol.

[10]As a concept of remote procedure calls, URL in an web-app is analogous to invoking a function on the host, so it can indicate certain transactions.

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 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.

V. Application Case Studies

In this section, we will present our experience in using Gyrus to protect existing applications. Our experiments cover both 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). As for web applications, we picked the following from the top 25 sites according to Alexa [1]: 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, and we base this argument on the Pew Internet survey called "What Internet Users Do On A Typical Day" [28], 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 involve 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.

A. 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 done in very intuitive way. 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, on 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.

B. 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-mails (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 will be clicked 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 clicking the `Send` button. On the event, Gyrus will extract user-intended texts from the "To", "Subject", and message body text panes.

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 on it. All user-generated text will be directly shown as the same text in the traffic, and Gyrus will extract each field to query to Authorization DB.

Constructing Signature We show the signature for this application in Example 1. 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 window, which is `ATH_Note` in this case. To improve the event condition that detects the application, we list all participating UI objects on user intent on `COND` part. Starting from the event receiving object, a `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 panes, the `CAPTURE` field will assign temporary variables `A`, `B`, and `C`, to each one. For the network monitor, we set protocol `TYPE` as `SMTP` and `METHOD` as `SEND`, and bind each of the variables assigned during the `CAPTURE` stage to protocol specific variables.

C. Digsby: Yahoo! Messenger & Twitter

Application Specification Digsby is a stand-alone client for accessing multiple instant messenger 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 messenger/online social network services can be easily 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. 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 compared with the authorization DB after 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.

D. Web Applications

There are some common characteristics among the web applications. Since we use Microsoft Internet Explorer 10 for running all web applications, window class name cannot be used to distinguish the application. Instead, we use the domain name of currently active page as an identifier. In addition, the protocol that the traffic is sent through is always HTTP(S) in web applications. Therefore, monitored transactions can be identified by associating a URL with a submit input event. Variables for matching user-intended text will be bound as HTTP parameters. In general, user-intended text in the packet is URL-encoded.

E. 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[11], along with To, Subject, and message panes.

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, only if it is set with URL parameter `act=sm`. User-intended text is transmitted on the `to`, `subject`, and `body` parameters of the POST request.

[11]We ran Gyrus with the old version of GMail composing UI which was available until July 2013.

Constructing Signature The `EVENT` field is set to `LCLICK`. For the application condition, the domain name of `mail.google.com` will be used as a window identifier, along with the relative positions of text boxes to the `Send` button. For network traffic, the trap condition is set to URL https://mail.google.com/mail with parameter condition `act=sm`, and variables in the `CAPTURE` field will be matched with POST parameters named `to`, `subject` and `body`.

F. Web-App: Facebook

Application Specification We focus on three transactions in the Facebook application: status update, adding comment, and sending message. For status update, a 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 message, a user presses the `ENTER` key after composing her message, which is analogous to the Digsby example.

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

Constructing Signature The event will be `LCLICK` for status updates, while it will be `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 comment, and link an icon name with "Add more friends to chat" and conversation history objects as siblings for sending a chat message.

G. Web-App: Paypal

Application Specification Using Gyrus with Paypal enables validation of the integrity of the amount of money sent to someone. On the transferring money page, a user clicks the button `Continue` after he types the username or e-mail address of the recipient, and the amount of money to transfer in the text boxes. The workflow is analogous to our e-mail examples, with the primary difference being that in this case the message is the amount of money to be transferred.

Network Traffic Specification After clicking the `Continue` button, traffic will be sent to https://www.paypal.com/us/cgi-bin/webscr with the POST parameter `cmd=_flow`. 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

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

TABLE I. LIST OF ACTIVITIES WHERE GYRUS CAN HELP TO PROTECT THE CORRESPONDING NETWORK TRANSACTIONS, FROM THE SURVEY 'WHAT INTERNET USERS DO ONLINE [29]', BY PEW RESEARCH CENTER.

https://www.paypal.com/us/cgi-bin/webscr, and its POST parameter named `cmd=_flow`. Finally, variables for captured text for amount of money and recipient will be linked to the POST parameters `amount` and `email`.

H. Discussion

In cases where the same protocol is used by multiple applications, 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 easily support new applications, 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* [25] from Microsoft) and some knowledge about the network protocol used by the application, but no further details about the internals of the application (as compared to if we used VM introspection techniques to extract user intent).

Although it is very 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 paper, 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.

In terms of 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 I shows the result of 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 I can be supported by Gyrus. Clearly, Gyrus can protect a large 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, we can use Access Control Gadgets [30] 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 to be unrealistic.

One limitation of Gyrus is that it cannot protect an application where 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 user-intended input, by running application logic along with these computation proof mechanisms. In addition, 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, 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 in dom-0 to 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, a point worth noting here is that 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, since for usability purposes, an application should not have too much variance in its UI, nor should it provide too many ways for performing the same operation; 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 we will continue to capture user intent correctly, unless the application changes its UI (which, again, for usability reasons, is less likely to happen).

VI. Evaluation

In this section, we present the results of our evaluation on the security, usability and performance of Gyrus when using it to protect the applications studied in Section V.

A. Security

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

1) 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 Sect. V. 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. All of the messages generated by this malware are blocked by Gyrus. 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.

2) 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 [20], [5], [40].

The next avenue for attack is against 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, thanks to the secure overlay, attackers are limited to misplacing user-generated, albeit unintended content in traffic allowed by Gyrus (e.g., switching the subject and content of an email, take users comment to one story on Facebook as his/her outgoing comment on another story), and thanks to our policy of only displaying on the overlay the content of the window which current have focus, the mistakenly sent out content must be from the "correct" application. Also, we believe 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 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.

B. Usability

From our experience of protecting the applications studied in Section V, 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 Section VI-C1 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 not only supports 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 [12], 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.

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

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

TABLE II. LATENCY INTRODUCED BY GYRUS WHILE PROCESSING THE INPUT. USER-INTERACTION DATA WAS COLLECTED DURING THE USE CASE EVALUATION.

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%)

TABLE III. NETWORK LATENCY FOR HTTP CONNECTION.

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.

C. 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. All the experiments presented in this section are performed on a commodity laptop: a Lenovo Thinkpad-T520, equipped with a dual-core Intel Core i5 2520m and 8GB of RAM. The dom-U runs 3 logical cores with 7GB of RAM, while dom-0 has 1 logical core and 1GB of RAM.

1) Interaction Overhead: In the worst case, on a system protected by Gyrus, the user will experience the following delay for every keyboard/mouse input: first, 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, secondly, 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 between 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 between 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. The results of our experiments are presented in Table II. To provide some context for interpreting the results, prior research suggests that acceptable range of such turn around time for interaction with human is 50-150ms [35]. Thus, our experiments show that on average case, users can smoothly interact with a system protected by Gyrus.

2) 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: 1) the time to establish an HTTP connection (and we used two test sites), 2) the time to load a web page with dynamic content, measured by The Chromium's Page Benchmark extension [37], 3) the effective bandwidth of a system, obtained by measuring the time to download a 550MB disk image from the Debian repository through HTTP. To measuring the overhead introduced by our Man-In-The-Middle (MITM) proxy for HTTPS connections, we did 2 tests: 1) download 15KB of web-page data from a public website, and 2) download a 32MB file from a remote HTTPS server. All experiments are repeated 10 times, and the average results are presented in Table III & IV.

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%)

TABLE IV. NETWORK LATENCY FOR HTTPS CONNECTION (WITH MAN-IN-THE-MIDDLE PROXY).

Comparing the results from a KVM Guest versus Gyrus running on it, Gyrus only introduces around 1ms of single response delay, less than 6% (53.8ms) 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, it incurs 4ms of delay on getting access to a single web-page data, and adds less than 6% of bandwidth overhead on downloading of file content. Evaluation results for the network latency shows that Gyrus has very little overhead, at worst 6% on both bandwidth and loading a webpage.

VII. FUTURE WORK AND CONCLUSIONS

There are many potentially fruitful areas for future work. The first is to simplify the process of supporting a new application by automating the analysis and generation of the UI and traffic signatures. Extending Gyrus' output monitoring to include disk transactions would allow Gyrus to support non-networked applications such as word processors. Integrating with a delegated computation verifier would allow Gyrus to support a broader range of applications. In addition, Gyrus could verify that the input to a computation verifier is actually from the user.

Another interesting future direction would be to implement Gyrus on other platforms. The current design can be adapted to work in a cloud computing model where the remote host is an instance in an IaaS cloud. For platforms where it is hard to deploy our current virtualization-based design (e.g., mobile devices), one could explore modifying the threat model to only defend against malicious applications, assuming that the underlying operating system is clean. Under this new threat model, it may be possible to achieve "what you see is what you send" by implementing a similar defense strategy as a component inside the platform's runtime framework (e.g., Android Dalvik).

To conclude, in this paper 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.

ACKNOWLEDGMENTS

The authors would like to thank to the anonymous reviewers, and our shepherd, Dongyan Xu, for their help and feedback. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

REFERENCES

[1] Alexa Internet. Alexa - Top Sites in United States. http://www.alexa.com/topsites/countries/US.

[2] Android Open Source Project. Dalvik Technical Information. https://source.android.com/tech/dalvik/index.html.

[3] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of np. *J. ACM*, 45(1):70–122, January 1998.

[4] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 483–501, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, New York, NY, USA, 2011. ACM.

[6] Weidong Cui, Randy H. Katz, and Wai tian Tan. Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers. In *Proc. of the Annual Computer Security Applications Conference*, 2005.

[7] J. Elson and A. Cerpa. RFC 3507 - Internet Content Adaptation Protocol (ICAP). http://www.ietf.org/rfc/rfc3507.txt.

[8] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[9] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

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

[13] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, February 1989.

[14] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 113–122, New York, NY, USA, 2008. ACM.

[15] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[16] Michael Hohmuth, Michael Peter, Hermann Hartig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors. In *Proc. of the ACM SIGOPS European Workshop*, 2004.

[17] IMSpector. IMSpector: Instant Messenger Proxy Service. http://www.imspector.org/wordpress/.

[18] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *Proc. of the ACM Conf. on Computer and Communications Security*, 2007.

[19] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2005.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[21] Mac OSX Developer Center. NSAccessibility Protocol Reference. https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/ApplicationKit/Protocols/NSAccessibility_Protocol/Reference/Reference.html.

[22] Stuart E. Madnick and John J. Donovan. Application and Analysis of The Virtual Machine Approach to Information System Security and Isolation. In *Proc of the Workshop on Virtual Computer Systems*, 1973.

[23] Lorenzo Martignoni, Pongsin Poosankam, Matei Zaharia, Jun Han, Stephen McCamant, Dawn Song, Vern Paxson, Adrian Perrig, Scott Shenker, and Ion Stoica. Cloud terminal: secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[24] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[25] Microsoft Developer Network. Inspect. http://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx.

[26] Microsoft Developer Network. UI Automation Overview. http://msdn.microsoft.com/en-us/library/ms747327.aspx.

[27] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

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

[29] Pew Internet. What Internet Users Do Online. http://www.pewinternet.org/Static-Pages/Trend-Data-(Adults)/Online-Activites-Total.aspx.

[30] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[31] Srinath Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[32] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *In USENIX Security*, 2012.

[33] Dave Shackleford. Blind as a Bat? Supporting Packet Decryption for Security Scanning. http://www.sans.org/reading_room/analysts_program/vss-BlindasaBat.pdf.

[34] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. 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, pages 121–130, New York, NY, USA, 2009. ACM.

[35] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, fourth edition, 2005.

[36] Steve Slaven. xautomation. http://hoopajoo.net/projects/xautomation.html.

[37] The Chromium Projects. Benchmarking Extension. http://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/chrome-benchmarking-extension.

[38] Stef Walter. Proxsmtp: An smtp filter. http://memberwebs.com/stef/software/proxsmtp/.

[39] Duane Wessels, Henrik Nordström, Alex Rousskov, Adrian Chadd, Robert Collins, Guido Serassio, Steven Wilton, and Chemolli Francesco. Squid: Optimising web delivery. http://www.squid-cache.org/.

[40] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.