Overhaul: Input-Driven Access Control for Better Privacy on Traditional Operating Systems

Kaan Onarlioglu, William Robertson, Engin Kirda
Northeastern University, Boston, USA
{onarliog,wkr,ek}@ccs.neu.edu

***Abstract*—The prevailing security model for OSes focuses on isolating users from each other; however, the changing computing landscape has led to the extension of traditional access control models for single-user devices. Modern OSes for mobile devices such as iOS and Android have taken the opportunity provided by these new platforms to introduce permission systems in which users can manage access to sensitive resources during application installation or runtime. One drawback of similar efforts on desktop environments is that applications must be rewritten with this security model in mind, which hinders traditional OSes from enjoying the benefits of user-driven access control.**

We present a novel architecture for retrofitting a dynamic, input-driven access control model into traditional OSes. In this model, access to privacy-sensitive resources is mediated based on the temporal proximity of user interactions to access requests, and requests are communicated back to the user via visual alerts. We present a prototype implementation and demonstrate how input-driven access control can be realized for resources such as the microphone, camera, clipboard, and screen contents. Our approach is transparent to applications and users, and incurs no discernible performance overhead.

I. Introduction

The prevailing security model for traditional operating systems focuses on protecting users from each other. For instance, the UNIX access control model provides a framework for isolating users from each other through a combination of user identifiers, group identifiers, and process-based protection domains. The fundamental assumption underlying this approach to security is that the primary threat to user data originates from other users of a shared computing system.

The traditional user-based security model makes sense in the context of timesharing systems, where many users share access to a common pool of computing resources. However, the modern proliferation of inexpensive and powerful computing devices has resulted in the common scenario where one user has sole access to a set of resources. Unfortunately, there exists a significant impedance mismatch between user-based access control and the primary security threat in the single-user scenario, where users inadvertently execute malicious programs that operate with that user's privilege and have full access to all of the user's sensitive computing resources. As such, user-based access control is not well-suited to preventing attacks against user confidentiality. In particular, malicious programs can access privacy-sensitive hardware devices such as the microphone or camera, or access virtual resources such as the system clipboard and display contents of other programs.

In response to the changing computing landscape, much effort has been invested in extending the user-based access control model to enable dynamic, *user-driven* security. For instance, modern operating systems for smartphone and tablet devices have taken the opportunity provided by these new platforms to introduce permission systems as an extension to the underlying UNIX security model that remains in use on these systems. For instance, iOS gives users the ability to approve or deny access to sensitive resources during runtime via popup prompts. Research operating systems have also proven a fertile milieu for experimenting with security models that address the needs of modern computing systems. For instance, Roesner et al. [27] present an extension to ServiceOS where gadgets are embedded into applications that allow users to grant or deny access to sensitive resources.

In each of the preceding examples, determining legitimate user intent and translating that intent into appropriate security policies is a central feature of their respective security models. For each system, security decisions as to whether to allow or deny access to sensitive resources for individual programs are delegated to the user, and the system is responsible for establishing trusted input and output paths to capture user intent such that malicious programs cannot influence this process by either spoofing or intercepting user inputs.

We fundamentally agree with this approach to securing modern computing devices, since users are often solely capable of classifying program actions as privacy violations or other inappropriate uses of their resources. However, one drawback of these efforts is that applications and operating systems must be written with this security model in mind. This requirement largely excludes traditional operating systems such as Windows, Linux, and OS X, which remain in wide use, from enjoying the benefits of user-driven access control. In this work, we show that providing a user-driven security model for protecting privacy-sensitive computing resources can be realized for traditional operating systems, as an extension to the traditional user-based security model. In particular, our security model is based on the observation that a legitimate application usually accesses privacy-sensitive devices immediately after the user interacts with that application (e.g., by clicking on a button to turn on the camera, or pressing the key combination for a copy & paste operation). We call this security model *input-driven access control*, and demonstrate how it can be enforced by correlating user input events with security-sensitive operations based on their temporal proximity, making access control policy decisions automatically based on this information, and notifying the user of resource accesses in an unintrusive manner. We achieve this by using lightweight and generic techniques to augment the operating system and display manager with trusted input and output paths, which we collectively call Overhaul, and demonstrate our approach by implementing a prototype for Linux and X Window System.

In contrast to prior work, we show that capturing user interaction as a basis for security decisions involving sensitive resources can be performed in an *application-transparent* manner, obviating the requirement that applications be rewritten to conform to special APIs or with a more refined security model in mind. Using our approach, we demonstrate how dynamic access control can be transparently achieved for common resources such as the microphone, camera, clipboard, and display contents. Finally, we show that this can be achieved without a discernible performance impact, and without utilizing intrusive prompts or other changes to the way users interact with traditional operating systems. To summarize, we make the following contributions.

- We present a general architecture for retrofitting a dynamic, input-driven access control model into traditional operating systems in a transparent manner, in which access to privacy-sensitive resources is mediated based on the temporal proximity of user interactions to access requests. We also address the challenges of tracking user interaction across process boundaries (e.g., IPC channels).
- We build upon this architecture to demonstrate how input-driven access control can be implemented to protect sensitive resources such as the microphone, camera, clipboard, and display contents.
- We present a prototype implementation for Linux and X Window System, and evaluate it to show that our system imposes no discernible performance overhead and no changes to the traditional computing interface.

II. Problem Statement

Security models for traditional operating systems center on multiplexed computation on timesharing systems, where multiple users share access to a single set of computing resources. However, the shift towards dedicated devices with single users has resulted in a fundamental impedance mismatch between the traditional model of users, groups, and processes and the needs of modern systems. In particular, contemporary threats often take the form of malicious programs that execute with the full privileges of the user, rendering user-based security models largely ineffective. Mobile operating systems such as iOS and Android, as well as research systems such as ServiceOS [27], have promoted the concept of *dynamic access control* where permissions to access sensitive resources are granted by users on-demand. However, operating systems for the desktop and server have been largely neglected by these advances, since prior work has required that applications be designed with dynamic access control in mind.

An open question remains as to whether modern dynamic access control can be realized for platforms where applications have *not* been written to conform to this model. We believe that our work answers this question in the affirmative.

Threat model. For this work, we assume that the trusted computing base includes the display manager, OS kernel, and underlying software and hardware stack. Therefore, we assume that these components of the system are free of malicious code, and that normal user-based access control prevents attackers from running malicious code with superuser privileges. On the other hand, we assume that the user can install and execute programs from arbitrary untrusted sources, and therefore, that malicious code can execute with the privileges of the user. We assume that complementary preventive security mechanisms are in place to prevent privilege escalation attacks, such as ASLR or DEP.

Input-driven access control primarily addresses two privacy breach scenarios. The first one covers programs that stealthily run in the background and access privacy-sensitive resources without the user's knowledge, behavior typical of malware [2], [3], [18], [7]. OVERHAUL ensures that such attempts are automatically blocked.

The second scenario involves benign, but buggy or misbehaving, applications that access protected resources without the user's knowledge. Due to the trade-offs OVERHAUL make in order to transparently retrofit a dynamic access control into existing systems, unlike previous work [27], it is not possible to match each input event to a precise user intent. Therefore, in this scenario, OVERHAUL instead visually notifies the user to alert her of the undesired resource access.

We note that all forms of user-driven security are fundamentally vulnerable to full mimicry attacks. For instance, if a user could be tricked into knowingly installing, executing, and granting privileges to a malicious application that imitates a well-known legitimate application, user-driven security models would fail to provide any protection. Hence, our threat model does not include this third scenario.

Goals. The primary security goals OVERHAUL aims to achieve are the following.

(S1) OVERHAUL must allow an application to access privacy-sensitive resources only if the user has explicitly interacted with that application through physical, hardware input devices, immediately before the access request. Resources include hardware devices such as cameras, microphones, and other sensors, or virtual resources such as system clipboards and the display contents of user programs.
(S2) OVERHAUL must prevent programs from forging input events or mimicking user interaction to escalate their (or other applications') privileges.
(S3) OVERHAUL must ensure that legitimate user interaction events cannot be hijacked by malicious applications, such that users should not mistakenly grant permissions to a malicious program that were intended for a legitimate program.
(S4) OVERHAUL must notify users of successful accesses to protected resources via a trusted output path that cannot be obscured or interfered with by other applications.

In addition to the above security properties, we set out to satisfy a number of design goals for OVERHAUL.

(D1) OVERHAUL must provide transparent protection to existing applications, without requiring access to source code or application modifications.
(D2) OVERHAUL should not incur a significant performance overhead.
(D3) OVERHAUL should not significantly degrade the usability of, or change the way users interface with the underlying system, for instance, by using intrusive popup prompts.

III. System Design

The architecture of an OVERHAUL-enhanced system requires modifications to, and close interaction between, several components of the operating system and display manager. In this section, we describe the abstract design of OVERHAUL, independent of the underlying operating system, and present the challenges involved in monitoring and tracking user input across process boundaries. Later, in Section IV, we will demonstrate how our design can be realized in a prototype running on Linux and the X Window Server.

Note that our work assumes a userspace display manager (i.e., a design similar to that of the X Window System), an approach employed by popular commodity operating systems. Different OS designs can allow display managers integrated into the kernel, which would alleviate the need for some of the components we describe below, such as a separate trusted communication channel between the kernel and the display manager. Our design can be applied to that case in a straightforward manner.

A. Trusted Input & Output Paths

In order to realize any of the aforementioned security guarantees, OVERHAUL must establish a trusted path for user input. By a trusted path, we refer to the property that input events should be authenticated as legitimately issued by a real user with a hardware input device, as opposed to synthetic input events that can be issued programmatically. This capability serves as a generally useful primitive that could be exposed to higher layers of the software stack. However, in this work we focus on illustrating its use for transparently securing access to system-wide resources.

The display manager of the system is often responsible for receiving all low-level input events, including mouse clicks and key presses, from device drivers and delivering them to their target application windows. Consequently, OVERHAUL utilizes a display manager with an enhanced input dispatching mechanism that can detect and filter out synthetically generated inputs to fulfill the trusted input path requirement.

Likewise, OVERHAUL is tasked with establishing a trusted output path to alert users whenever a sensitive resource access request is granted. We achieve this through visual notifications that appear on the screen. Since the display manager is in control of the screen contents, OVERHAUL extends it with an overlay notification mechanism that is always stacked on top of the screen contents, and cannot be obscured, interrupted, or interfered with by other processes.

B. Permission Adjustments

The kernel is responsible for dynamically adjusting the privilege level of user programs in response to permission granting actions, i.e., authentic user input events. In order to accomplish this task, the kernel first needs to establish a secure communication channel to the display manager. The display manager can then use this channel to send the kernel *interaction notifications* each time the user interacts with an application. Since the display manager is often a regular userspace process, the kernel is able to authenticate the communication endpoint and ignore communication attempts by other processes in a straightforward manner.

The kernel keeps a history of these interaction notifications, which include the identity of the application that received the interaction and a timestamp, inside a *permission monitor*. Once this information is stored, the permission monitor can respond to permission queries and adjustment requests, originating either from the userspace display manager through the already established secure communication channel, or from within the kernel, any time a permission decision is to be made. This decision process involves comparing a timestamp issued together with the query with the stored interaction timestamp corresponding to the target application, and in this way correlating privileged operations with input events based on their temporal proximity.

Finally, the kernel also uses the secure communication channel to request from the display manager that it display a visual alert when a resource access is granted.

C. Sensitive Resource Protection

An important class of system resources that OVERHAUL aims to protect is sensitive hardware devices. These devices could include arbitrary sensors attached to the system; typical examples on desktop operating systems include the camera and microphone. In order to implement dynamic access control over hardware resources, the kernel is responsible for mediating accesses to these sensitive hardware devices.

However, note that the kernel does not interpose on all security-sensitive resources. Representative examples include the system clipboard and program display contents, both controlled by the display manager. Applying dynamic access control over these resources requires the display manager to query the kernel permission monitor, and grant or deny the action based on the response.

To illustrate the enhancements required to the kernel and the display manager, and how sensitive resources are protected, we present two scenarios that build upon the components described above. For the following, we let:

$$\begin{aligned}
\mathsf{op}_t &\ \text{be a privileged operation at time } t, \\
&\ \text{where } \mathsf{op} \in \{\mathsf{copy}, \mathsf{paste}, \mathsf{scr}, \mathsf{mic}, \mathsf{cam}\}, \\
E_{A,t} &\ \text{be an input event sent to application } A \text{ at time } t, \\
N_{A,t} &\ \text{be an interaction notification corresponding to } E_{A,t} \\
Q_{A,t} &\ \text{be a permission query for application } A \text{ at time } t, \\
R_{A,t} &\ \text{be a response } \in \{\mathsf{grant}, \mathsf{deny}\} \text{ for } Q_{A,t}, \\
V_{A,op} &\ \text{be a visual alert request, indicating } A \text{ performs } \mathsf{op}.
\end{aligned}$$

Hardware resources. Figure 1 presents an example interaction involving an application's request to access the system microphone. In an unmodified system, the request would succeed so long as application A holds the permission to access the microphone device at $t + n$.

OVERHAUL introduces the following changes. First, the system ensures that for all applications the permission to access the microphone is denied by default. (1) When the user clicks on a button in application A to turn on the microphone at time t, the display manager receives the input event $E_{A,t}$ and verifies that it is generated by a hardware input device through user interaction. (2) If $E_{A,t}$ is authentic, then the display manager first sends the kernel permission monitor an interaction

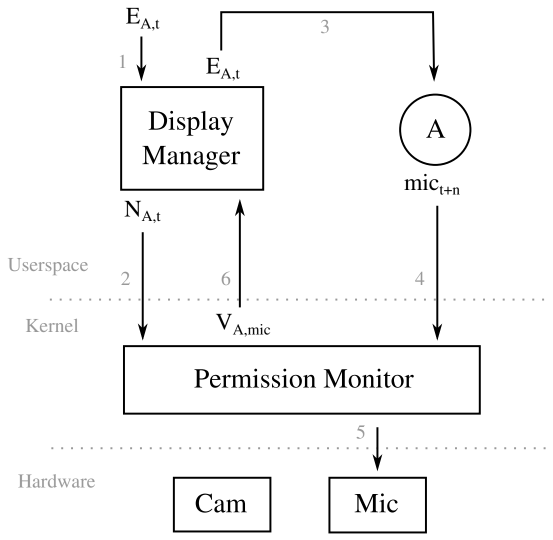

Figure 1. Dynamic access control over privacy-sensitive hardware devices.

notification $N_{A,t}$ through the secure communication channel. The permission monitor records this notification, indicating that A received authentic user input at t. (3) The display manager then forwards $E_{A,t}$ to its destination A. (4) Upon receiving the event, A attempts to turn on the microphone. The permission monitor intercepts A's request mic_{t+n} to access the device. It compares A's latest interaction time t with the device access request time $t+n$ to correlate the input event with the privileged operation, based on a preconfigured threshold δ. (5) Access to the device is granted to A only if the privileged operation could successfully be correlated with a preceding input event (i.e., if $(t+n)-t=n<\delta$ holds). (6) Finally, the kernel sends $V_{A,mic}$ to the display manager to request that the user be alerted. This step is necessary because the display manager may not have adequate information to identify the process that actually accessed the resource (e.g., due to IPC mechanisms, as explained in Section III-D).

The verification of user input authenticity provides the property that sensitive device access operations can only be performed in response to legitimate user input. Note that, in this scenario, no permission query from the display manager to the permission monitor is necessary. Since the kernel has full mediation over hardware resources, the permission monitor can implicitly adjust the permissions of A when necessary. This entire process is transparent to the application.

Display resources. Figure 2 shows an example interaction for a clipboard paste operation between the display manager and an application A. The baseline protocol consists of A requesting the clipboard contents from the display manager, and receiving back the copied data. OVERHAUL revokes all clipboard access permissions by default, and modifies the protocol in the following way.

(1) First the user inputs the keystrokes to paste some text, (2) the display manager verifies that the input $E_{A,t}$ is authentic and notifies the kernel permission monitor with $N_{A,t}$, (3) and forwards the key event to A. (4) After receiving the command from the user, A issues a clipboard paste request paste_{t+n} to the display manager. (5) Instead of immediately serving the request, the display manager sends a permission

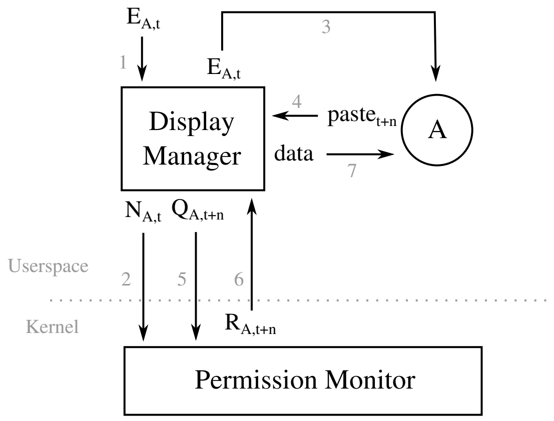

Figure 2. Protecting copy & paste operations against clipboard sniffing.

query $Q_{A,t+n}$ to the kernel permission monitor through the secure communication channel. (6) As before, the permission monitor compares the interaction time t in its records for A with the privileged operation request time $t+n$ issued together with the query. If the correlation of the input event with the operation request is successful based on the temporal proximity threshold δ, (i.e. $n<\delta$), the permission monitor replies with a grant response $R_{A,t+n}$; otherwise $R_{A,t+n}$ is a deny response. (7) If and only if $R_{A,t+n}$ is a permission grant does the display manager return to A the data; or else A is blocked from accessing the clipboard. In this scenario an explicit visual alert request from the kernel is not necessary, because the display manager can successfully identify the requesting process without kernel assistance.

Here, the secure communication channel between the kernel and the display manager is used both for sending interaction notifications to the permission monitor, and for querying it whether to allow the privileged operation.

As before, the verification of user input authenticity provides the property that copy & paste operations can only be performed in response to actual inputs. This provides protection against malicious programs that attempt to capture sensitive data from the system clipboard, such as passwords pasted from a password manager. We note that because permission queries are implicitly generated along with the copy & paste requests, this protection is transparent to the application.

Note that, in this scenario, first sending input notifications to the permission monitor and later querying it for the same information could seem unnecessary. Instead, one could store input notifications inside the display manager to avoid kernel communication. However, in the next section, we show that our design is necessary for the kernel to track interactions across process boundaries, through process spawns and IPC channels.

D. Interaction Across Process Boundaries

Real-life applications often consist of multiple processes or threads, and communicate with each other using application-specific protocols via inter-process communication (IPC) facilities provided by the OS. This significantly complicates the task of associating user input with privileged operations requested by an application, because the process receiving the input event could be different from the actual process

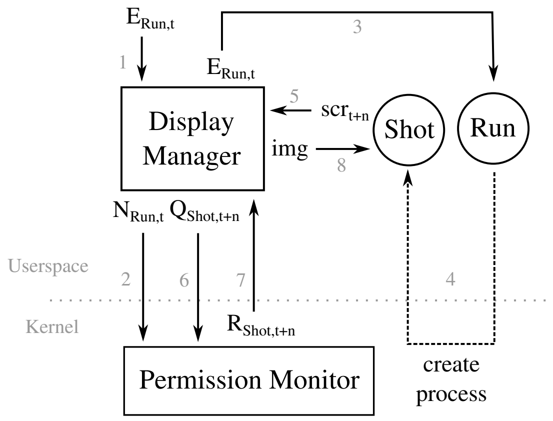

Figure 3. A program launcher executing a screen capture program, illustrating the need for interposing on process spawn mechanisms to propagate interaction information.

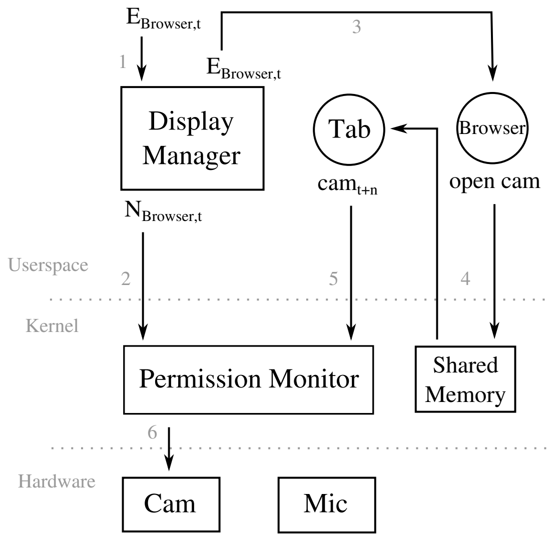

Figure 4. A multi-process browser, components of which communicate via shared memory IPC. This example illustrates the need for interposing on IPC endpoints to propagate interaction information.

that accesses a sensitive resource. We illustrate this challenge OVERHAUL needs to address with the examples below.

Figure 3 presents a scenario where an application Shot attempts to capture a screen image. Since the screen content is also a resource controlled by the display manager, this example is similar to the previous copy & paste example. However, here, the user first executes a program launcher Run, types in the name of the program Shot, and the application launcher executes Shot on the user's behalf. In other words, (1–3) the user actually interacts with Run, which the kernel permission monitor records; (4) but Run creates a new process Shot, (5) and the screen capture request scr_{t+n} is made by this different process for which there exists no interaction record.

In another scenario, Figure 4 depicts how a multi-process Internet browser that uses separate processes for each browser tab (i.e., similar to Chromium) would run a web-based video conferencing application. (1–3) When the user commands the browser to launch a video conference session, she actually interacts with the main browser window Browser, and the permission monitor is notified of this. However, Browser opens the web application in a separate process Tab and (4) commands it to turn on the camera via shared memory IPC. As a result, (5) Tab requests cam_{t+n} without a corresponding interaction record in the permission monitor.

The ubiquity of multi-process application architectures, applications that launch third-party programs, and IPC use make it necessary for OVERHAUL to correctly handle cases similar to those exemplified above. Therefore, our design requires OVERHAUL to interpose on all process and thread spawns, as well as the entire range of IPC mechanisms provided by the OS (e.g., (4) in Figure 3 and Figure 4). Specifically, OVERHAUL needs to propagate interaction notifications between processes according to the following policy:

(P1) Interaction notifications of a parent process must be propagated to a newly spawned child process; i.e., whenever a process X creates a new process Y, all interaction notifications $N_{X,t}$ recorded in the permission monitor must be duplicated as $N_{Y,t}$.

(P2) In an IPC channel established between two (or more) processes, interaction notifications of a message sender process must be propagated to the receiver process; i.e., OVERHAUL must monitor all established IPC endpoints, and whenever process X sends a message to process Y, interaction notifications $N_{X,t}$ recorded in the permission monitor must be duplicated as $N_{Y,t}$.

In this way, OVERHAUL can support process spawns and IPC chains of arbitrary length and complexity, and remain transparent to the applications and oblivious to the application-level communication protocols.

E. Limitations

OVERHAUL inherently shares the limitations of other user-driven security approaches. In particular, because the user's perception of malice and their interaction with applications are central to this security model, OVERHAUL cannot provide protection against malware that can trick users into voluntarily installing and using it, for example, by mimicking the appearance and functionality of well-known legitimate applications. Additionally, OVERHAUL does not support running scheduled tasks, or persistent non-interactive programs that need access to the protected sensitive devices (e.g., a cron job or daemon that periodically takes screen captures). We stress that these issues are fundamental to any user-driven access control model, and despite its limitations OVERHAUL provides important security benefits complementing the standard access control models employed in commodity operating systems, without any significant detriments to performance or user experience.

The trade-offs OVERHAUL makes between backwards compatibility with legacy programs and defending against on-system malware results in a system that provides strictly weaker security guarantees than prior work on user-driven access control [27], where a stronger connection between user intent and program behavior can be achieved. This primarily stems from the design decision to treat existing applications in a black-box fashion. Nevertheless, we believe that OVERHAUL significantly raises the bar for attacks, bringing much of the security of user-driven access control to existing platforms in a transparent manner.

IV. Implementation

In this section, we focus on the implementation details of how we guarantee the properties required of each component of OVERHAUL. Though our design is sufficiently general to apply to OSes that implement a traditional access control model, we built our prototype for the Linux kernel and X.Org implementation of the X Window System.

A. Enhancements to X Window System

The X Window System is responsible for enforcing several security properties outlined in Sections II & III. In particular, it must guarantee that a trusted path exists for authentic user input, a trusted output for visual notifications, and interpose on all accesses to the display contents and system clipboard.

Trusted input. The underlying assumption behind our prototype implementation of a trusted input path for the X Window System is that user inputs that originate from hardware attached to the system should be considered authentic, while software-generated events should be untrusted. While there are legitimate use cases for allowing programmatic generation of input events (e.g., GUI testing tools) such avenues are also required for malware to interact with a user interface on the user's behalf so long as the hardware is considered to be free of embedded malicious functionality.

As a result, OVERHAUL focuses on distinguishing between hardware and software-generated input events. We identified two facilities provided by X11 for generating and injecting synthetic events to the event queue: the `SendEvent` [30] and XTestFakeInput [15] requests. `SendEvent` is a core X11 protocol request that allows a client to send events to other clients connected to an X server. In particular, this interface could allow malware to inject keystrokes or mouse events on other windows. However, events sent using this interface must have a flag set that indicates that the event is synthetic. As such, filtering such input events within the X server is a matter of checking for the presence of this flag.

The second request, XTestFakeInput, is part of the XTest extension, which is used to provide a GUI testing framework. In this case, it is not possible to implement a flag check since no indicator flag is used with XTest requests. Therefore, it was necessary to modify the X server to tag events with the extension or driver that generated the event. While this is more onerous than checking for the existence of a flag, it is also a method for determining the provenance of input events that generalizes to future modifications to the X Window System.

With the ability to distinguish hardware-generated input from synthetic input, the X server was modified to connect to a secure communication channel upon initialization (as we will explain in Section IV-B), and send interaction notifications to the kernel permission monitor every time the user interacts with an X client. These notifications are labeled with the PID of the process that received the event and a timestamp. The PID serves as an unforgeable binding between a window belonging to a process and events, as the mapping between X client sockets and the PID is retrieved from the kernel.

We note that the trusted input path described so far remains vulnerable to clickjacking attacks [20]. For instance, a malicious X client may place transparent overlays on the screen, or periodically display a previously invisible window over other applications in an attempt to trick users into clicking on them and stealing authentic input events. To prevent this, OVERHAUL only generates interaction notifications if the X client receiving the event has a valid mapped window that has stayed visible above a predefined time threshold.

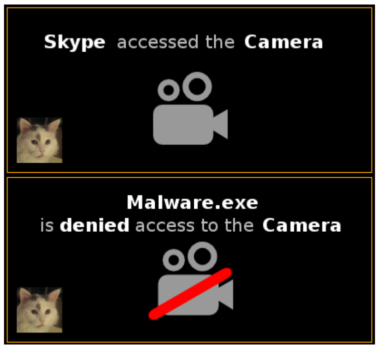

Figure 5. Sample visual alerts shown by OVERHAUL. The cat image is used as the visual shared secret to indicate that the alert is authentic.

Trusted output. As described before, the trusted output path that OVERHAUL utilizes is a visual alert shown on the screen whenever a sensitive resource is accessed. Since the X Window System controls the entire display contents, OVERHAUL ensures that the displayed alert is rendered on top of all other windows, and cannot be blocked, obscured, or manipulated by other X clients. We have designed the alert messages to be displayed for a few seconds at the top of the screen at a reasonably large size to be easily noticeable. Since resource accesses can only be granted immediately following user input, the user is highly likely to be present and interacting with the computer, making it difficult for her to miss an alert. In addition, the alerts make use of a visual shared secret set by the user of the system to prevent malicious applications from forging fake alerts. Two example alerts are shown in Figure 5.

Note that, compared to popup prompts that require explicit policy decisions from the user during runtime (e.g., Windows User Account Control, or iOS permission dialogues), alerting the users with visual notifications inherently establishes a looser association between user actions and the application behavior. Indeed, we have implemented and verified that OVERHAUL's security primitives can be used to support such a security model in a trivial manner, where the trusted output path would be used for displaying an unforgeable prompt, and the trusted input path to verify user interaction with it. However, it has been shown that popup prompts have severe usability issues that conflict with their security properties, and that they are often ignored by users, or disabled completely [24]. Therefore, we believe the non-intrusive, transparent approach we have taken with OVERHAUL is a worthwhile trade-off between security and usability, and would be a more effective security solution in a real-life setting. We do not explore the popup prompt approach further in this paper.

Display contents. The X Window System allows any client program to access the contents of the root window (i.e., the entire screen), or any specific window through the `GetImage` core protocol request [30], or the `XShmGetImage` request provided by the MIT shared memory extension [11]. These

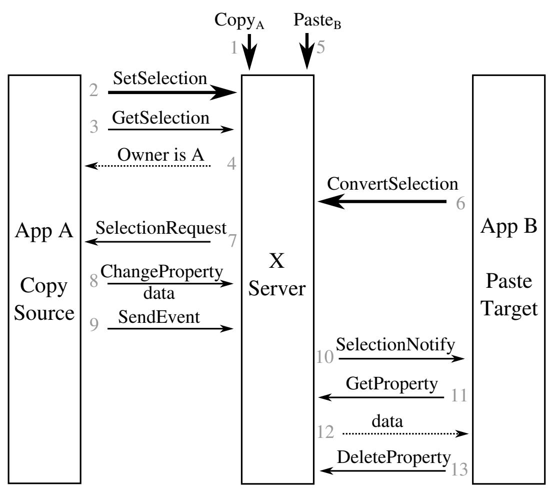

Figure 6. Protocol diagram for the X11 copy & paste operation. Modified steps are highlighted in bold.

interfaces can be used to retrieve the displayed contents for any purpose, such as taking screenshots, or recording the desktop.

In order to mediate accesses to the display contents of X clients, our modified X server intercepts these events, and queries the kernel permission monitor via the secure communication channel with a message containing the PID of the requesting process and a timestamp. Based on the response, access is either granted, or the screen capture request is dropped. This way, OVERHAUL can enforce that display contents can only be accessed in response to user input.

The X Window System also provides two additional core protocol requests, `CopyArea` and `CopyPlane`, which are used for copying a representation of display contents between two buffer areas. These requests could be used as an alternative approach to capture the screen contents, and therefore, OVERHAUL must also interpose on them. However, unlike the previous `GetImage`, these requests are not specifically designed for capturing display contents, and they are regularly used by X clients for various other purposes. Therefore, in this case, OVERHAUL first needs to inspect the owners of the source and destination buffers specified in the copy request. If the owners of both buffers are identical, in other words, a client is copying a portion of its own window, the request is allowed to proceed. However, if a client is requesting the display contents owned by a different client (or the root window), OVERHAUL applies its user input-based access control as before, and allows or blocks the request accordingly.

Clipboard. The X Window System does not provide a central clipboard space, but instead defines the copy & paste operations as an inter-client communication protocol [28] outlined in Figure 6. The steps to copy data from a *source client* to a *target client* are as follows.

(1) A copy operation is initiated by user input received via an X input driver. (2) The source client asserts ownership of a *selection* object by issuing to the X server a `SetSelection` request. In (3) and (4) the source client confirms with the X server that it has successfully acquired the selection. This concludes the copy operation; note that no data has actually been copied at this stage.

(5) The paste event is initiated by user input. (6) The target client sends a `ConvertSelection` request to the X server, (7) which, in turn, issues a `Selection Request` to the selection owner (i.e., the source client) to notify it of the request for the copied data. (8) The source client sends the data to the X server to be stored as a *property* using a ChangeProperty request, (9) and then requests from the server that the target client be sent a `Selection Notify` event, using a `SendEvent` request. (10) The paste target is notified that the copied data is available. (11) The target client responds with a `GetProperty` request, (12) retrieves the data, (13) and finally, removes it from the server.

In Figure 6, the protocol steps that were modified in OVERHAUL are highlighted in bold. In particular, steps (1) and (5) are events that are verified as authentic user input from a hardware input device. The X server notifies the kernel permission monitor of these events as previously described. In steps (2) and (6), before serving the `SetSelection` or `ConvertSelection` requests received from the clients, the X server first queries the kernel permission monitor via the secure communication channel to confirm that the copy or paste request is preceded by corresponding user interaction. The operation is allowed to proceed only if the permission monitor responds with a permission grant message; otherwise, the client is sent back a *bad access* error.

Note that, this copy & paste protocol is followed merely by convention, and the given interaction sequence is not enforced by the X server. As a result, a malicious X client may attempt to skip certain steps of the protocol to bypass OVERHAUL's checks. One possible attack vector is the `SendEvent` request which allows an X client to command the X server to send an X11 event on behalf of the client. By exploiting this mechanism, a malicious client can directly send `SelectionRequest` events to other clients and receive the copied data from the selection owner. To prevent such attacks, our implementation also interposes on the `SendEvent` requests, and blocks the sending of events that can break the copy & paste protocol. Other examples of possible attacks include subscribing to events generated by the X server when properties are created and updated to retrieve the pasted data stored in them before the actual paste target could remove it. OVERHAUL ensures that such events are only delivered to the paste target while the clipboard data is in flight. Due to space restrictions, we omit details of these low-level implementation details.

B. Enhancements to the Linux Kernel

As shown in Section III, our implementation augments the Linux kernel with a permission monitor that establishes a secure communication link to the X Window System, mediates sensitive hardware accesses, adjusts per-application privileges in response to interaction notifications, and responds to permission queries from the X server for access to display resources.

Secure communication channel. The first property that our kernel must support is establishing and authenticating the communication channel to the X Server. In our prototype, we used the Linux netlink facility to provide this channel [29]. Netlink was originally designed to exchange networking information between the kernel and userspace, but it serves as a robust general communication channel across this boundary.

Netlink, however, does not solve the authentication problem. That is, the kernel and X server must ensure that no malicious program is interposing on the channel. While using a standard mutual authentication protocol is possible, our prototype instead relies on the fact that the kernel operates in supervisor mode and can introspect on the userspace X process. Once the kernel establishes the netlink channel and receives a connection request from X during server initialization, it examines the virtual memory maps to check whether the process it is communicating with is indeed the X server. In particular, it checks whether the executable code mapped into the process is loaded from the well-known, and superuser-owned, filesystem path for the X binaries. If so, it considers the remote party to be authenticated as the legitimate X server and, due to the kernel's supervisor privileges, the X server trusts that the kernel will perform this procedure correctly.

Device mediation. OVERHAUL must interpose on all accesses to sensitive hardware devices. To this end, it suffices on Linux to monitor `open` system call invocations on device nodes exposed in the filesystem. Therefore, our prototype implements an augmented `open` system call that, in addition to normal UNIX access control checks, looks up the interaction notification records received from the X server for the running process to allow or deny access to the device accordingly. Note that it is usually considered better practice to implement kernel-side security checks using the Linux Security Modules (LSM) framework [32], instead of modifying system calls directly. However, as of this writing, LSM does not officially support stacking multiple security modules. Since OVERHAUL is not a replacement for other security modules, we implemented our prototype in this way as a conscious design choice.

An important implementation detail of our prototype deals with accurately mapping sensitive devices to their filesystem paths. In particular, modern Linux distributions often make use of dynamic device name assignments at runtime using frameworks such as udev. Therefore, our prototype relies on a trusted helper application, owned by the superuser and protected against unauthorized modification using normal user-based access control, to manage this mapping. It is invoked in response to changes in the device filesystem, mounted by convention at `/dev`, and propagates these changes to the kernel via an authenticated netlink channel.

Process permission management. The kernel permission monitor receives interaction notifications from the X server, which includes a PID and a timestamp, and needs to record this information in an easily accessible context associated with each process. Our prototype stores this information inside the `task_struct`, which is the data structure Linux uses to represent a process. Every `task_struct` is implicitly associated with a unique process; therefore, this procedure only requires us to locate the structure corresponding to the PID reported inside the interaction notification, and save the interaction timestamp inside a new field.

To perform a permission check, the permission monitor first receives the PID of the process that requests access to the sensitive resource, either internally from the device mediation layer, or from the X server via the netlink channel. Next, it retrieves the correct `task_struct` and compares the timestamp recorded there (i.e., the most recent user interaction time) with the privileged operation's timestamp. If the temporal proximity of the two is above a configurable threshold, permission is granted (or a positive response is sent back to the X server). We empirically determined that setting a threshold of less than 1 second could lead to falsely revoked permissions, but 2 seconds is sufficient to prevent incorrectly denying access to legitimate processes. In our long-term experiments with this configuration, described in Section V-D, we did not encounter any broken functionality or unusual program behavior.

Process creation and IPC. As previously explained, OVERHAUL must be able to track interaction information across process boundaries for any meaningful real-life use. In Linux, a new process (i.e., the child) is created by duplicating an existing process (i.e., the parent), using the `fork` or `clone` system calls. This operation duplicates the `task_struct` of the parent to be used for the child process, which includes the interaction timestamp stored in the same data structure. In other words, our implementation ensures that the parent's interaction information is passed down to a newly created child automatically, without additional modification to the kernel. This property also extends to the threads of a process, because Linux does not have a strict distinction between processes and threads and uses a separate `task_struct` for each.

In contrast, tracking interaction information across IPC channels requires further modifications to the kernel, for each IPC facility provided by the OS. Our implementation supports all of POSIX shared memory and message queues, UNIX SysV shared memory and message queues, FIFOs, anonymous pipes, and UNIX domain sockets. Higher-level IPC mechanisms that are built on these OS primitives (e.g., D-Bus) are also automatically covered. These IPC mechanisms are modified in a similar manner to propagate interaction information between the two endpoint processes, which works as follows.

(1) When an IPC channel is first established, we embed inside the kernel data structures that correspond to the IPC resource an expired interaction timestamp. (2) When a process wants to send data through an IPC link, it first embeds inside the IPC resource its own interaction timestamp, unless the structure already contains a more recent timestamp. (3) When the receiving process reads the data from the channel, it compares its own interaction timestamp with that is embedded inside the IPC resource. If the IPC channel has a more up-to-date timestamp, the process saves it in its `task_struct`.

Implementation of this protocol requires adding a timestamp field inside the IPC data structures, and inserting checks inside the corresponding send and receive functions for each IPC facility. However, a notable exception is POSIX and SysV shared memory, which must be handled differently. Specifically, once the kernel allocates and maps a shared memory region with the `mmap` system call, writes and reads to these regions are regular memory operations that cannot be intercepted above the hardware level. We overcome this obstacle by taking a different approach. We interpose on virtual memory mapping operations inside the kernel, check whether the mapped area is flagged as *shared* (indicated by a flag inside the corresponding `vm_area_struct`), and if so, revoke read and write permissions for that memory area. This causes subsequent accesses to that memory region to generate access violations, which allows OVERHAUL to capture the IPC attempt inside the page fault handler. We then run the interaction propagation protocol described above, and temporarily restore the memory access permissions to their original values to allow the memory operation to succeed on the next try. Clearly, repeating this process for every memory access could lead to severe performance overhead; therefore, after every access violation, we put the corresponding `vm_area_struct` on a wait list before its permissions are revoked once again. This allows memory accesses that immediately follow the first page fault to proceed uninterrupted. This wait duration must be sufficiently shorter than the 2 second interaction expiration time, since we would miss shared memory IPC attempts and fail to propagate interaction timestamps during this period. We configured this duration to 500 ms, which yielded a good performance-usability trade-off as shown in Section V.

CLI interactions. A final implementation requirement arises from the fact that Linux systems often make extensive use of the command line interface. On graphical desktops, this is achieved by running a terminal emulator (e.g., xterm) which communicates with a command line shell (e.g., bash) via a pair of *pseudo terminal* devices. If the user was to type in the name of a command line application inside a terminal emulator (as opposed to using a graphical application launcher), the terminal emulator would receive the input events, and communicate the command to launch to the shell via the pseudo terminal devices. Any subsequent device access requests would be made by a program launched by the shell process, which has not received any direct interaction (In fact, the shell usually is not even an X client and, thus, cannot receive X11 input events).

To enable command line tools that access the protected sensitive devices to function correctly under OVERHAUL, we implemented an interaction timestamp propagation protocol analogous to the one described for IPC channels above. Here, the modifications are made inside the pseudo terminal device driver. Whenever a process writes to a terminal endpoint, that process embeds its timestamp into the kernel data structure representing the pseudo terminal device. Subsequently, when another process reads from the corresponding terminal endpoint, that process copies the embedded timestamp to its `task_struct`, unless it already has a more recent timestamp.

Processes isolation and introspection. OVERHAUL does not require sandboxing of individual user applications, or any advanced process isolation mechanism beyond the kernel and process memory isolation that commodity operating systems provide. In particular, all interaction notifications in our design are managed by the OS; they are never exposed to userspace applications. This prevents malicious applications from tampering with legitimate interaction notifications to mount denial-of-service attacks, or hijacking interaction notifications of other processes. Similarly, since each interaction notification is bound to a specific process, malicious applications that run in the background and receive no user interaction cannot hijack the permissions granted to another application.

However, process introspection and debugging facilities offered by OSes need attention, because they might make it possible to inject malicious code into legitimate applications that are expected to have access to sensitive resources. In Linux, this threat is somewhat contained since the Linux debugging facilities, such as `ptrace` and `/dev/{PID}/mem` (also using `ptrace` internally), do not allow attaching to processes that are not direct descendants of the debugging process. In other words, even if two unrelated processes run with identical (but non-super user) credentials, they cannot manipulate each other's state. In our implementation, we provide even stricter security by temporarily disabling all permissions for a debugged process, with a trivial patch to the `ptrace` system call. This also prevents parent processes from tracing their own children, which, in turn, subverts attacks where a malicious program could launch another legitimate executable, and then inject code into it. OVERHAUL enables this protection by default, but it could be toggled by the super user through a `proc` filesystem node to facilitate legitimate debugging tasks.

Table I. PERFORMANCE OVERHEAD OF OVERHAUL.

Benchmarks	**Baseline**	**OVERHAUL**	**Overhead**
Device Access	45.20 s	46.18 s	2.17 %
Clipboard	116.48 s	119.93 s	2.96 %
Screen Capture	68.26 s	69.86 s	2.34 %
Shared Memory	234.86 s	236.33 s	0.63 %
Bonnie++	47319 files/s	47265 files/s	0.11 %

V. EVALUATION

A. Performance Measurements

Since OVERHAUL is an input-driven system that only impacts the operations performed on privacy-sensitive resources, we expect its performance overhead to be overshadowed by human-reaction times and I/O processing delays. Indeed, in our experiments with the prototype implementation, we did not observe a discernible performance drop compared to normal system operation. Consequently, in order to obtain measurable performance indicators to characterize the overhead of OVERHAUL, we created micro-benchmarks that exercise the critical performance paths of our system. We also used a standard filesystem benchmarking utility to measure the impact of our modified `open` system call on regular filesystem operations. We explain each of these benchmarks in more detail below.

Device access. In this benchmark, we measured the time to open the filesystem device node corresponding to the microphone installed on our testing system 10 million times.

Clipboard operations. We designed this benchmark to measure the runtime for performing 100,000 clipboard operations. Since in the X Window System a paste is significantly more costly than a copy, we configured our benchmark to only perform pastes for this test, and report the worst-case results.

Screen capture. This benchmark takes 1,000 screen captures using the `imlib2` library and measures the total runtime. The time to save the image files to disk is not included.

Shared memory IPC. Although OVERHAUL interposes on every IPC mechanism, our preliminary measurements indicated that the shared memory communication incurred the highest overhead due to the necessity for intercepting page faults, changing virtual memory access permissions, and invalidating page tables. Consequently, to measure the worst-case performance impact, in this benchmark we measured the runtime for performing 10 billion write operations on a shared memory area. We repeated this benchmark with different shared memory sizes (i.e., from 1 to 10,000 pages, with a page size of 4096 KB), and experimented with sequential and random write patterns. We found no correlation between these parameters and the performance impact; the overhead was near-identical in all runs. Here, we present the results for a shared memory size of 10,000 pages, and random writes.

Filesystem. To measure the performance impact of OVERHAUL on regular filesystem operations, we ran Bonnie++ [1], configured to create, stat and delete 102,400 empty files in a single directory. Since OVERHAUL does not interpose on `stat` or `unlink` system calls, we were unable to reliably measure any overhead for stat and delete operations, as expected. Therefore, we only report the runtime overhead for file creation.

For the purpose of this evaluation we temporarily modified OVERHAUL's permission monitor to grant access to resources even when there is no user interaction, in order to exercise the entire execution path of the benchmarked operation. We repeated all tests on a Linux system with OVERHAUL, and on a system with an unmodified kernel and X server, five times each, and compared the average results when calculating the overhead. Experiments were performed on a computer with an Intel i7-930 processor, 9 GB memory, and running Arch Linux x86-64. We present the results of our experiments in Table I.

Our measurements show that OVERHAUL performs efficiently, with the highest overhead observed being below 3%. Note that these experiments artificially stress each operation under unusual workloads, and the overhead for a single operation is on the order of milliseconds in the worst case, and ranging down to below a nanosecond. Hence, the overhead is often not noticeable by the user. Moreover, the Bonnie++ benchmark demonstrates that OVERHAUL does not significantly impact the performance of regular file open operations.

B. Usability Experiments

We conducted a user study with 46 participants to test the usability of OVERHAUL. The participants were computer science students at the authors' institution, recruited by asking for volunteers to help test a "defensive security system". In order to avoid the effects of priming, participants were not informed about the functionality of OVERHAUL. The only recruitment requirement was that the participants are familiar with using Skype and web browsing, so that they could perform the given tasks correctly. No personal information was collected from the participants at any point.

The participants were asked to perform two tasks to test different aspects of our system. The first task presented them with a Skype instance on our test machine running OVERHAUL, logged into a test account. They were asked to perform a call to a second test account, while OVERHAUL performed its security checks without their knowledge. Once complete, an experimenter asked the participants to compare this process with their previous experience of using Skype. Specifically, they were asked to rate the difficulty involved in interacting with the test setup on a 5-point Likert scale, where a score of 1 indicated that their experience was almost identical, and 5 indicated that the test setup posed significant difficulty.

In the next task, the participants were asked to perform a specific search on the Internet on an OVERHAUL-enabled machine. While they were occupied with the task, a hidden background process that attempted to access the camera was triggered at a random time, which was blocked by OVERHAUL and caused a visual alert to be displayed. Once the task was complete, the participants were asked to explain whether they have noticed anything unusual while performing their tasks.

At the end of the first phase of the experiment, all 46 participants found the experience to be identical to using Skype on an unmodified system. This empirically confirms that OVERHAUL is transparent to the users. In the second phase, 24 participants immediately interrupted the task when the OVERHAUL notification was displayed, and alerted the experiment observer to the blocked camera access. Another 16 noticed the alert, however continued the task and reported the unexpected camera activity after being prompted by the observer. Only 6 users reported not having noticed anything unusual. These results confirm that OVERHAUL alerts are able to draw most users' attention while they are occupied with other tasks, and are effective security notifications.

C. Applicability & False Positives Assessment

To understand whether OVERHAUL interferes with the normal functionality of applications, or produces false alerts due to incorrectly blocked legitimate programs, we tested the system on common applications. To compile the application pool for this task, we first manually inspected the descriptions of all *Top Rated* packages in the Ubuntu Software Center, and identified those that access the resources OVERHAUL is designed to protect. Next, we searched the official and community package repositories of Arch Linux, our experiment environment, with relevant keywords (e.g., webcam, microphone, screenshot, capture, record), and added the hits to the pool. After eliminating the packages that do not work (e.g., due to missing dependencies) we ended up with 58 applications consisting of video conferencing tools (e.g., Skype, Jitsi), audio/video editors (e.g., Audacity, Kwave), audio/video recorders (Cheese, ZArt), screenshot utilities (Shutter, GNOME Screenshot), and screencasting tools (e.g., Istanbul, recordMyDesktop). The pool also included popular web browsers (e.g., Firefox, Chrome); in those cases we tested them with various web-based video chat applications. Note that the application pool contained both GUI and console programs. We manually experimented with each application to verify that they work as expected, observed whether OVERHAUL alerts were displayed correctly, and whether there were false alarms.

In our experiments, we encountered a single application that produced what could be considered a spurious alert. Specifically, we observed that Skype attempted to access the camera as soon as the program was launched, before the user logs into the application. When Skype was configured to automatically start on boot, this situation led to a camera access without user interaction, and consequently, OVERHAUL blocked the access and produced an alert. This did not cause subsequent video calls to fail, and we argue that blocking such unanticipated device accesses is the desired behavior in order to achieve OVERHAUL's security properties.

While we did not encounter any malfunctioning application, this experiment also revealed a peculiar limitation of OVERHAUL. Specifically, some of the screenshot tools we tested included an option to delay the shot by a user-specified time. By design, OVERHAUL does not support this functionality since the interaction notifications associated with the application expire before the screen could be captured.

For testing OVERHAUL's clipboard protection mechanism we used an additional set of 50 applications including popular office programs, text and media editors, web browsers, email clients, and terminal emulators. Since OVERHAUL does not display alerts for clipboard accesses due to usability reasons, we instead verified correct functionality by inspecting the logs produced by our system. In these tests we did not encounter any false positives or incorrect program behavior.

We note that OVERHAUL does not support running scheduled tasks, or persistent non-interactive programs that access the protected devices (e.g., a cron job that periodically takes screenshots). While we did not encounter such applications in our tests, this remains a fundamental limitation of our system.

D. Empirical Experiments

Due to ethical concerns, and the necessity of installing a custom kernel and malware samples on users' machines, it is a difficult task to design a large-scale user study to test the long-term security and usability properties of OVERHAUL. Therefore, one of the authors instead volunteered to experiment with OVERHAUL on their personal home and work computers.

For this experiment, we implemented a sample malware that runs in the background during the computer's normal operation and spies on the user. In particular, it periodically retrieves clipboard contents, takes screenshots, and records sound samples from the microphone. For privacy reasons, our sample did not record camera images. Since the test was performed on actual, personal machines used on a daily basis, we only stored the captured information on disk, while real malware would exfiltrate it to a remote site. We stress that our malware sample was created to mimic the behavior of real information-stealing malware [2], [3], [18], [7], exploiting the standard interfaces to the sensitive resources exposed by the OS. No functionality was artificially added or removed that would ease its detection. We installed this malware on two different computers belonging to one of the authors, who was made aware of the collected information, and volunteered for the task. We enabled OVERHAUL on one of the machines, while the other was left running unmodified, without protection. We left the malware running for 21 days. Both computers were actively used everyday for work and personal use.

At the end of the experiment we confirmed that the malware running on the OVERHAUL-protected system could not collect any information, as expected. We checked OVERHAUL's logs and verified that attempts to access the protected resources were detected and blocked. The malware on the vulnerable computer, on the other hand, was able to successfully spy on the user. We manually investigated the collected data and found sensitive information including screenshots of bank account information displayed on an e-banking site, and email exchanges. The data sampled from the clipboard included passwords copied from the password manager, phone numbers, and excerpts from emails. The malware was also able to collect voice recordings from the headset microphone. We also investigated OVERHAUL's logs to see which applications were granted access to the protected resources. The camera and microphone were used by two video conferencing applications. Screen was captured by the system's default screenshot tool, and by a desktop recording application. Clipboard accesses were logged for a large number of applications. During the testing period of 21 days, we did not encounter any cases of legitimate applications being incorrectly blocked.

These observations show that spying malware can be severely damaging, and that OVERHAUL is effective at improving user privacy in the face of attacks. Conducting a similar long-term study at a larger scale, in a more scientific framework, is a difficult yet promising future research direction.

VI. RELATED WORK

Previous work has studied capturing user intent to implement user-driven access control. Roesner et al. [27] present an approach in which permission granting is built into user interactions with permission-granting GUI elements called access control gadgets (ACG). The authors extend ServiceOS to provide this capability to application developers, and require that applications be modified to use ACGs. This work captures user intent at a fine granularity and provides stronger security guarantees than OVERHAUL as each action is precisely mapped to a permission. However, our goal is to propose an architecture that can be retrofitted into traditional OSes transparently. In our work, we encountered a different set of challenges stemming from the fact that we are dealing with traditional systems (i.e., Linux) that do not provide the features that ServiceOS does.

Gyrus [21] is a virtualization-based system that displays editable UI field entries in text-based networked applications back to the user through a trusted output channel, and guarantees that this is the information sent over the network. BLADE [22] infers the authenticity of browser-based file downloads based on user behavior. While sharing similar goals with OVERHAUL, these address different security problems.

Systems that use timing information to capture user intent include BINDER [13] and Not-a-Bot [19]. BINDER associates outbound network connections with input events to build a host-based IDS. However, its design does not address the challenges of IPC, making it unsuitable for use with certain applications that OVERHAUL targets. Not-a-Bot uses TPM-backed attestations to tag user-generated network traffic on the host, and a verifier on the server that checks them to implement DDoS, spam and clickjacking mitigation measures. These systems target network-based attacks, whereas OVERHAUL aims to control access to privacy-sensitive devices.

Some systems that advocate user-authentic gestures for secure copy & paste between domains are the EROS Window System (EWS) [31], Qubes OS [6], and Tahoma [12]. Similarly, in this work, we also address the problem of secure copy & paste so that malicious applications cannot intercept these requests. There has also been much work in the domain of trusted computing. For example, Terra [17], Overshadow [10], and vTPM [9] use virtual machine technology for enabling trusted computing. In contrast to the above, OVERHAUL does not require use of virtualization, or explicit user cooperation.

Several operating systems and applications employ popup prompts to defer privacy policy decisions to users [5], [4], [8]. However, this approach to user-driven access control has been shown to suffer from usability issues; for instance, Motiee et al. [24] demonstrate that Windows users often find User Account Control prompts distracting, dismiss them without due diligence, or disable them completely. OVERHAUL sidesteps these concerns by taking a transparent, unintrusive approach. Flash Player employs a mechanism that only allows clipboard operations initiated by user input [23]. OVERHAUL generalizes this application-specific defense to the entire system and other sensitive resources, and provides the additional security property that user input cannot be generated synthetically.

Quire [14] is an extension to Android that enables applications to propagate call chain context to downstream callees. Hence, applications can verify the sources of user interactions, and make policy decisions accordingly. There has also been much work that aims to enforce install time application permissions within Android (e.g., Kirin [16], Saint [26], Apex [25]). These approaches enable the user to define policies for protecting themselves against malicious applications. OVERHAUL is orthogonal to the smartphone platform security work.

VII. CONCLUSIONS

This paper has shown that an input-driven access control model based on enforcing the temporal proximity of user interactions to an application's sensitive-resource access requests can be retrofitted into traditional operating systems. We have presented an abstract design independent of the underlying OS, and described our implementation for Linux and X Window System. Our approach fulfills the design goals enumerated in Section II. OVERHAUL provides a trusted input path between the user and kernel, a display manager that authenticates hardware-generated input events and interposes on display resources, and a kernel permission monitor that mediates access to sensitive hardware (S1), (S2). The display manager also enforces appropriate visibility requirements on application windows to prevent hijacking of authentic user interaction (S3), and ensures that resource accesses are communicated to the user via visual alerts (S4). OVERHAUL requires no modifications to existing software, and is transparent to users (D1), (D3). The performance evaluation and empirical tests of our prototype show that it remains efficient and practical, while increasing the security of traditional operating systems (D2).

In future work, we plan to investigate gray-box approaches to input-driven access control that close the gap between white-box approaches [27] that require applications to be written with user-driven access control and the black-box approach adopted here. One promising direction is to leverage static and dynamic program analyses to more precisely link user intent, user input, and device accesses, all without requiring modifications to existing programs.

Acknowledgments. This work was supported by the National Science Foundation (NSF) under grant CNS-1409738, and Secure Business Austria.

REFERENCES

[1] "Bonnie++," http://www.coker.com.au/bonnie++/.

[2] "CERT Polska - Slave, Banatrix and Ransomware," http://www.cert.pl/news/10358.

[3] "Dell SonicWALL Security Center - Malware switches users Bank Account Number with that of the attacker," https://www.mysonicwall.com/sonicalert/searchresults.aspx?ev=article&id=614.

[4] "Flash Player Help - Privacy settings," http://www.macromedia.com/support/documentation/en/flashplayer/help/help09.html.

[5] "OS X Mountain Lion: Prompted for access to contacts when opening an application," http://support.apple.com/en-us/HT202531.

[6] "The Qubes OS Project," http://www.qubes-os.org/trac.

[7] "Trojan-Spy:W32/Zbot," http://www.f-secure.com/v-descs/trojan-spy_w32_zbot.shtml.

[8] "Windows Help - What is User Account Control?" http://windows.microsoft.com/en-us/windows/what-is-user-account-control.

[9] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *USENIX Security*, 2006.

[10] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems," *SIGOPS OSR*, vol. 42, no. 2, Mar. 2008.

[11] J. Corbet, "MIT-SHM (The MIT Shared Memory Extension)," http://www.x.org/releases/X11R7.7/doc/xextproto/shm.html.

[12] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A Safety-Oriented Platform for Web Applications," in *IEEE S&P*, 2006.

[13] W. Cui, R. H. Katz, and W. Tan, "Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers," in *ACSAC*, 2005.

[14] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *USENIX Security*, 2011.

[15] K. Drake, "XTEST Extension Protocol," http://www.x.org/releases/X11R7.7/doc/xextproto/xtest.html.

[16] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *ACM CCS*, Nov. 2009.

[17] T. Garfinkel, J. C. B. Pfaff, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-based Platform for Trusted Computing," in *ACM SOSP*, Oct. 2003.

[18] A. Gostev, "The Flame: Questions and Answers," http://securelist.com/blog/incidents/34344/the-flame-questions-and-answers-51/.

[19] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks," in *USENIX NSDI*, 2009.

[20] L. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, "Clickjacking: Attacks and defenses," in *USENIX Security*, 2012.

[21] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A Framework for User-Intent Monitoring of Text-Based Networked Applications," in *NDSS*, 2014.

[22] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "BLADE: An Attack-agnostic Approach for Preventing Drive-by Malware Infections," in *ACM CCS*, 2010.

[23] I. Melven, "User-initiated action requirements in Flash Player 10," http://www.adobe.com/devnet/flashplayer/articles/fplayer10_uia_requirements.html.

[24] S. Motiee, K. Hawkey, and K. Beznosov, "Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices," in *SOUPS*, 2010.

[25] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *ASIACCS*, 2010.

[26] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-centric Security in Android," in *ACSAC*, Dec. 2009.

[27] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems," in *IEEE S&P*, May 2012.

[28] D. Rosenthal, "Inter-Client Communication Conventions Manual," http://www.x.org/releases/X11R7.7/doc/xorg-docs/icccm/icccm.html.

[29] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov, "Linux Netlink as an IP Services Protocol," http://www.ietf.org/rfc/rfc3549.txt, Internet Engineering Task Force, Jul. 2003.

[30] R. W. Scheifler, "X Window System Protocol," http://www.x.org/releases/X11R7.7/doc/xproto/x11protocol.html.

[31] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, "Design of the EROS Trusted Window System," in *USENIX Security*, 2004.

[32] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security*, 2002.