Towards Understanding Systems Through User Interactions

by

Doran R. Smestad

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2015

APPROVED:

Professor Craig A. Shue, Major Thesis Advisor

Professor Krishna K. Venkatasubramanian, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

Modern computer systems are complex. Even in the best of conditions, it can be difficult to understand the behavior of the system and identify why certain actions are occurring. Existing systems attempt to provide insight by reviewing the effects of actions on the system and estimating their cause. As computer systems are strongly driven by actions of the user, we propose an approach to identify processes which have interacted with the user and provide data to which system behaviors were caused by the user. We implement three sensors within the graphical user interface capable of extracting the necessary information to identify these processes. We show our instrumentation is effective in characterizing applications with an on-screen presence, and provide data towards the determination of user intentions. We prove that our method for obtaining the information from the user interface can be done in an efficient manner with minimal overheads.

Acknowledgements

While a ‘thesis’ is defined as an effort of individual research, the simple fact is that I would not be here today without the support of others.

I would like to thank my advisor, Professor Craig Shue, for all of his time and patience with me as I worked through the research for this thesis. His support and feedback were invaluable throughout the course of my work, both during the times of success and the times of ‘constructive criticism.’ Working with him was a great privilege, and without him I would never have been able to pursue the master’s degree.

I would like to thank my parents for always pushing me to be my best in all aspects of life. Their support in all my endeavors gave me the confidence to attempt new things and seek out opportunities. Their insistence and expectation of excellence drove me to fight through the difficult problems and emerge triumphant. Mom, Dad, thank you.

I thank my grandfather, Alan Sock, for the inspiration to become a computer scientist. Even at a young age I remember seeing his enthusiasm for all things electronic, and have always valued each and every conversation we have. His enthusiasm and willingness to allow my experimentation with the various computers, laptops, and other devices around his house drove me to learn more, and then pursue a formal education in the field.

I send a collective thanks to all my peers in the Applied Logic and Security lab for their feedback on my work and keeping the long days of work entertaining. In particular, Doug, thank you for always getting me to smile, even if you didn’t mean to, on the worst of days.

I thank the faculty and fellow students in the Applied Logic and Security, and Performance Evaluation of Distributed Systems, research groups for the friendly research atmosphere and their feedback on my thesis work.

I thank my reader, Krishna K. Venkatasubramanian, for his time and feedback on my thesis work.

Finally, I would like to thank my best friend and fiancée, Haley, for supporting me during the entire course of my master’s degree. Her unconditional love, friendship, and strength provided me with the will to succeed. Her constant reminders to eat, accompanied with her excellent cooking, kept me alive throughout the year. I look forward to spending the rest of our lives together.

This material is based upon work supported by the National Science Foundation under Grant No. 1422180. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

Contents

1 Introduction

Modern computer systems have advanced to the point where the cause of activity on a host is not always clear. Existing security and audit systems leverage limited information in attempts to describe and judge the legitimacy of system behavior without knowing if the observed actions are intended by the user. These systems focus only on logged behavior in well defined spaces, for example: process execution stats, network traffic, and file accesses. However, as computer systems are driven by the user, these system should also consider the actions of the end-user. Our approach instruments the graphical user interface (GUI) to determine which processes interact with the user and to extract the context of the interaction towards an understanding of user-endorsed actions on a system.

Existing approaches try to gain an understanding of actions occurring on a system by focusing on the symptoms of the actions and attempt to work backward. These approaches have studied system call data to determine the application's intended purpose is. Other systems take this approach further and attempt to determine if an application is acting maliciously through patterns of system calls and heuristics of file accesses [1, 2].

These approaches have mixed success as they have no data on what the user's actions were. A similar approach attempts to incorporate the user somewhat indirectly by approving network connections only within a certain time window after detecting activity on the keyboard [3]. Another approach specifically looks for an open or save file dialog box prior to allowing a process to edit files [4]. All these approaches fall short of understanding the user interaction with processes and as such attempt to use just parts of the interaction to provide marginal benefits. Our approach aims to record the full interaction between a user and system processes.

We place our focus on processes and how they interact with the user. With modern systems being driven by user, the user will be the root of many actions occurring on a system. We aim to capture the interactions of the user with the system by instrumenting the graphical user interface. All processes wishing to communicate with the user must appear on the screen, and in order to receive commands from the user they must be able to accept data from the keyboard and mouse. In order to have a screen-presence processes must engage existing libraries and subsystems to convert text and graphical commands into objects displayable on the shared resource of a screen. Further, an application must also utilize system interfaces to be notified of incoming user input from the keyboard and mouse. We take advantage of these requirements and place instrumentation within the interfaces a process must interact with to achieve user interaction.

By collecting and storing data from the GUI we are able to provide data towards a number of interesting applications. For example, our data collection could be employed for widely deployed usability testing, similar to how applications such as Firefox collect data on how users interact with their windows [5], our instrumentation could provide insights into how users navigate the entire operating system. Our instrumentation could also be used towards the auditing of user activity and which applications a user engages with. This could be particular useful for organizations investigating reports of misuse, or simply for the gathering of application usage metrics to decide if it is worth renewing licenses for expensive programs. Additionally, our data could be leveraged towards providing extra insight into a system's operations when evaluating a security policy. In particular, gathered user interaction data could be sent to a network controller when determining if an application should be permitted to contact a server holding confidential data.

We implement our approach in the Ubuntu Linux distribution and as such frame our discussions around the details of the Ubuntu platform [6]. Our decision to use the Linux-based platform was due to the open-source model of the Linux kernel, the Ubuntu platform, and all of the libraries included in the distribution's packaged release. Additionally, building our approach within the open-source platform would not be hindered by unavailable APIs or licensing restrictions. Despite our decision, we do note that our general approach and data gathering is applicable to other operating systems.

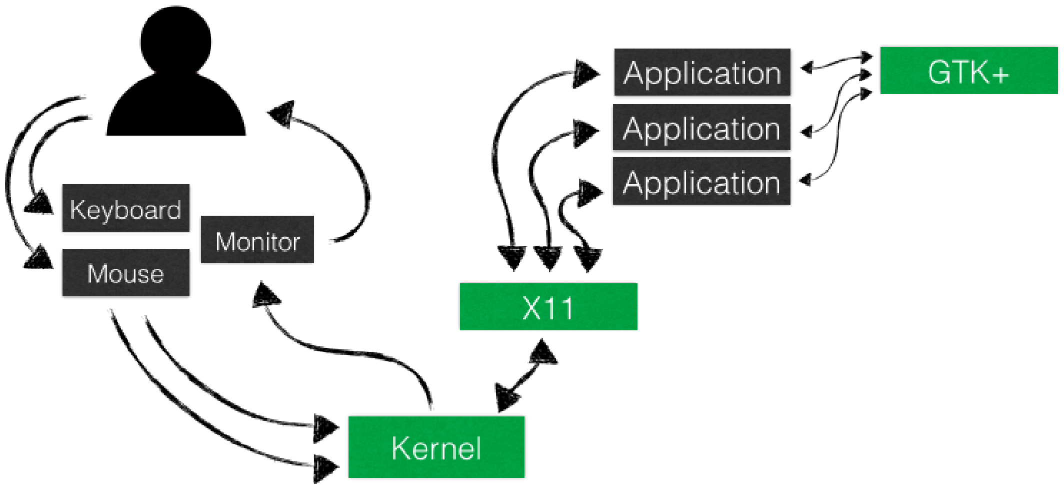

Figure 1: A high-level view of the information flow to and from the user. The three sensors, labeled Kernel, X11, and GTK+, will be placed within these components handling the flow of information allowing the extraction of data.

To extract the information necessary to make these distinctions, we created a three-sensor system as visualized in Figure 1. Each of the three sensors is located within a key portion of Ubuntu's graphics processing stack. We place a sensor within the kernel (the "kernel sensor"), another built into the X.Org implementation of the X Window System (the "X11 sensor") [7], and the final sensor within the untrusted user space graphics library GTK+ (the "GTK+ sensor") [8]. These three discrete sensors will hold differing levels of detail, all of which are required for our approach: the kernel records the reading of input devices, the windowing system tracks which applications appear on the screen and receive data; and the graphics toolkits render text on the screen. At each stage in the procedure, to either display an object on the screen or deliver user input a process, our sensors will log the extracted information and archive the data to disk. In all, we monitor user keyboard and mouse input, observe which programs receive user input, note applications which appear on the screen, and record contextual information for displayed processes. Furthermore, as the graphics toolkits reside fully under the end-user's control, we will verify and correlate data extracted across all three sensors to ensure consistency.

In addition to our own instrumentation, we also leverage the Linux Auditing Subsystem to record the actions the kernel performed on behalf of running processes. The Linux Auditing Subsystem is a set of functions within the kernel capable of tracking a number of different events typically employed to meet audit compliance regulations [9]. For our purposes we request the Auditing Subsystem to record the system calls invoked by each process that is not owned by root. With the audit data in hand we aim to link high-level user actions to the low level results; for example, when the user presses a button we observe the process perform some series of system calls in response.

In summary, the contributions of this thesis are:

- **High Performance Monitoring of the GUI:** We create and evaluate three high performance and low-overhead sensors capable of monitoring the entire graphical interface. Our sensors do not cause noticeable delay in a system and are designed for deployment across an organization.
- **Identification of Processes Reading Input Devices:** Through the creation of our own kernel module, we monitor requests from processes to receive keyboard and mouse data. If the request is approved, we record the amount of data each process receives from the user input devices.
- **Metrics for Process-User Interaction:** Through small modifications to the X Windowing System, we can detemine which processes have a screen presence, the quantity of information transmitted to the display, the amount of user input a process receives, and even which keyboard characters were received by the process.
- **Application independent context extraction:** Through instrumentation in the GIMP Toolkit (GTK+) graphics library, we extract both text and widget hierarchy information representing the graphical state of applications with a screen presence. By altering the GTK+ library, we are able to provide this context without changes to applications while efficiently extracting information with high accuracy.
- **A method for verification of an untrusted sensor:** By trusting the kernel and display manager sensors, we will explore overlaps to verify the data recorded by the GTK+ sensor. We enumerate both the indicators of valid recorded data and indicators of incorrect data.
- **A characterization towards identifying user-initiated behavior:** Based on our gathered sensor data, combined with audit logs from the Linux Auditing Subsystem, we propose characterizations for the identification of user-initiated process behavior.

We now introduce each sensor in greater detail and discuss the data each sensor is responsible for extracting.

The Kernel Sensor is implemented as a kernel module and resides within kernel space. By intercepting the `open`, `read`, and `close` system calls we are able to monitor each processes' use of the input devices. For example, when a user moves the mouse or types on the keyboard, the kernel will register data flowing in from the hardware and provide the information to any process which has opened the input devices for reading. By tracking which applications have opened the devices, and by logging the number of bytes read from each device, our kernel sensor is able to indicate which processes are receiving the raw user input. We estimate that the most frequent consumer of the raw data will be the windowing system.

The X11 Sensor is strategically placed within the code of the X Windowing System. Responsible for multiplexing the screen, the X Windowing System's Server (abbreviated "XServer") communicates with connected client processes (abbreviated "XClients") to process display requests and deliver events. The XServer handles the placement of on-screen windows, updating of displayable objects, and dispatching input events to the appropriate XClients. We place instrumentation within the XServer to log both the delivery of user-input events, and the application requests for displaying information on the screen. Unfortunately, these display requests are usually already rendered images and, as such, it is not possible to extract further information about what the process is displaying.

The GTK+ Sensor is an untrusted sensor, since it is instrumented within the user-controlled graphics library GTK+. This library, although is ultimately owned by root, is executed under the end-user's permissions. This raising the possibility of either data manipulation or outright lying to trick our sensor into logging information which is irrelevant. Despite this possibility though, we instrumented the graphical library to gain access to the rich contextual information available. For example, when an application based upon GTK+ creates a new text object, we are able to log the text to a file. Similarly, when a process creates a new button, we are able to log the parent-child relationship and pull out the child's text for further review. Once a process feeds GTK+ the information it wants displayed, GTK+ rendered the graphical objects and presents an image ready to be processed by the X Windowing System.

Using these sensors, we aim to study how data being reported from each sensor relates to one another while recording actions from a process. In particular, how information from the X11 and Kernel sensors relate to the data reported by the GTK+ sensor. For instance, when a user types into an editable text box displayed on the screen, the pressed keys will be detected by the kernel sensor, processed by the X11 sensor, and then the same text will often be logged as rendered on the screen by the GTK+ sensor.

2 Background and Related Work

In this section we discuss both recent best practices in computing and work related to our approach. We focus on work aimed at understanding actions on the system and work utilizing user interactions to provide legitimacy to observed actions.

2.1 Least User Privileges

Similar to the compartmentalization of confidential information, organizations often deploy user workstations with a minimal set of allowed user permissions. By restricting users to the least amount of access possible the overall integrity of the system can remain intact, even if a user account is compromised. This approach to security is commonly referred to as "least user privileges" [10]. This is typically deployed so that users do not have administrative access to their own machines and therefore cannot install new programs, alter the system configuration, or otherwise elevate themselves to 'root-level' permissions. In doing so, the operating system can be protected against malware infections and the occasional careless action by a user. Under the conditions of this model, users also cannot circumvent any administratively-applied constraints or monitoring systems such as our graphical interface sensors.

2.2 Detecting Abnormal Behavior

Work related to ours focuses on detecting anomalous behavior and attempting to determine if the abnormal actions are malicious in nature. The Panorama system created by Yin *et al.* constructs a virtualized testing host with system-wide sensitive data tainting to run suspect programs. If sensitive data is handled inappropriately (e.g. a program attempts to send it out a network socket) Panorama marks the program malicious [11]. Similarly, Siren runs suspect programs by emulating an environment in which there is an active user performing routine tasks. Malware which attempts to blend in with normal behavior can be detected by noticing deviations between test runs. Both of these approaches rely on dedicated testing environments, our approach aims to be deployed on an end-user system and perform dynamic identification of suspect behaviors [12].

Other previous work took a different approach, one which was more dynamic and run on the end-host itself. The work by Hofmeyr et. al., was able to detect differences in system calls patterns between benign programs and ones with malicious intent [2]. However, a newer research paper, AccessMiner, claims that looking solely at patterns of system calls cannot reliably detect malware. Instead, AccessMiner logged system calls from nearly a dozen lab systems and studied read/write attempts to detect malicious programs operating outside of normal file access patterns [1]. While quite successful with malware which perform file access, AccessMiner detection capabilities is restricted to malware that attempts to alter files normally in other program's system files. Furthermore, both of these papers can only attempt to detect patterns without any idea of what the intended end goal is; our approach will provide data towards attempting to infer the user's goal by how he/she interacts with on-screen applications.

Similarly, user-profiling approaches study user action as seen through the effects of their interaction with a system. Approaches in this area have attempted to use features like event frequency, event patterns, and the duration of interaction [1, 2, 13]. The creators of such approaches have studied these user effects through a number of means, such as Markov Chains, Na´ive Bayes classifiers, and Support Vector Machines [14–19]. Instead of looking at the after-effects of user actions, our approach studies the interaction directly as seen by the graphical user interface. We aim to provide the information gathered to the analytic models of the related works for more accurate models of user activities.

2.3 Inferring User Intentions

In order to learn of the user's intention while the user is working towards some goal, certain systems have proposed that either the operating system, the GUI manager, or even both be redesigned to support extra windows asking for a confirmation when dealing with sensitive operations. Perhaps the most widely experienced occurrence of this approach are the security confirmations for modern iOS or Android cell phone platforms when an application wishes to use certain components, such as the location or camera subsystems [20, 21]. A number of previous works describe how a similar system could be created for end-host systems, either by some sort of dialog box [22], or by having certain system-controlled user interface icons that allow access when activated by a user [23]. These approaches are beneficial, but require the cooperation of all user-facing applications and are therefore difficult to implement on a large scale. Our approach is to perform system changes without requiring any alterations to client applications.

Approaches most similar to ours are passive observation techniques which monitor the state of the GUI and record input received from the user to aid in security decisions. Notably, the work by Cui *et al.* receives raw mouse/keyboard events and uses this information to only allow outgoing network connections from processes recently provided input by the user [3]. Shirely *et al.*, takes this approach a bit further and applies it to standard filesystem tasks as well; for example, if the user interacts with a save-file dialog, allow the process to write to the file [4]. Another work simply captured an image of the user's screen and applied optical character recognition (OCR) to determine the contents of the interface [24, 25]. Our approach aims to get even more detail by extracting all displayed text from the GUI and correlate this data with system actions to infer the user's intent.

2.4 Detecting User-Interaction

A number of works have created systems varying in complexity to detect the interaction of a user. Some approaches simply detect generic events within the kernel from the input devices (e.g., keyboard, mouse) and leverage the presence of such an event to validate high-level actions, such as a network request [26]. Other approaches focus on specific programs, such as Firefox, and observe user-interactions and subsequent actions [27,28]. Further approaches perform both kernel-level and application-level logging to validate actions [29–31]. While these approaches many overlap with small portions of our approach, we aim to provide more detail than these approaches combined through our three-sensor instrumentation of the graphical interface.

2.5 Correlation of Sensors at Separate Trust Levels

There have been many approaches to correlating, aggregating, and detecting abnormalities in collected data in previous work. Some work needing only simple extraction from system call logs relied on regular expressions [4], while other work used a n-gram approach to create buckets of incoming system log data for further processing [1]. Work attempting to detect masquerading have used Bayes classifiers [15,16] and also have applied Support Vector Machines (SVMs) [17,19]. Further work have used Markov models, or simply have customized their own solution to best fit their data and approach [2,3]. Our approach will indicate ares of overlap between our trusted sensors and untrusted sensors to provide data for the application of these works to detect possible misinformation.

3 Trust Model

Our approach will assume least user privileges, which means an end-user has no administrator or root-level capabilities [10], and that neither the kernel nor root-level applications can be compromised by an adversary. We will further assume that there are no successful privilege escalation attacks by an unprivileged user to obtain root-level permissions. We will extend this to mean that anything running in the kernel, or with root permissions is inherently trusted, while all other applications are not. We assume that there are no resource exhaustion attacks launched, both in terms of a network-based denial of service, and also host-based exhaustion attacks like filling the hard drive with data. Finally, we also assume that the user is trustworthy, i.e. there are no insider threats. We do allow for any code to be run under user permissions, whether it be run legitimately by the user, or malware crafted by some adversary.

We explicitly place the kernel within our trusted computing base. With the kernel being the most heavily protected part of the operating system, we believe this assumption of integrity reasonable. This assumption is also made by anti-virus and similar modern security solutions. While other approaches, such as virtual machine introspection [32] could aid in relaxing this assumption, we leave such an approach to future work.

4 First Sensor: Kernel

The first of our sensors for identifying user interactions with processes is our kernel sensor. By monitoring and extracting information from communications between processes and the kernel, we can determine which processes have read input from any attached user-input device (e.g. mouse, keyboard, tablet). We store all information to a log for processing in realtime or for later auditing. Our sensor is implemented as a loadable kernel module, which has minimal overhead both in terms of memory space and computational performance.

4.1 Approach

Our first sensor is the kernel sensor. The kernel is the core part of the modern operating system. It is responsible for creating the layer of abstraction between the raw hardware and applications which wish to engage in its use. It handles the allocation and distribution of computer resources, such as processing time and memory space, and creates a consistent API and environment for processes each time they are started. The kernel is responsible for securing the system; therefore we must explicitly trust the kernel in our approach. Any program wishing to interact with the hardware must first engage the kernel; as such, our approach instruments the kernel to record the requests to access the input devices.

When programming on modern operating systems, there are a set of "system calls". These calls are particular procedures invoked by applications, often dealing with protected or shared resources, which will 'call' the system to execute kernel-level code to fulfill a request. These are required so that the kernel can enforce security policies, ensure no two processes attempt to edit the same resource at the same time, and provide a seamless layer of abstraction over vastly different hardware. For example, if a process would like to open a file, it is required to invoke the `open` system call which will validate the requesting process has permissions to access the file, ensure the file is not already in use, and allocate resources for the correct hardware to interact with the file. Our kernel sensor leverages these necessity of these system calls to track which processes have requested information from the input-devices.

To be effective, our kernel sensor's code needs to reside within the bounds of the kernel's protected space. To achieve this we wrote a loadable kernel module capable of performing the logging our approach requires. While the same goal can be obtained through modification of the kernel code itself, we decided on a kernel module for the ease of programming and to allow our approach to easily used on any machine running a current version of the Linux kernel. Our module intercepts three system calls vital to the use of the input devices: `open`, `read`, and `close`. Upon each intercepted invocation of the `open` system call we review the parameters of the call for file paths dealing with the input devices (e.g. `/dev/input/`), and log the request. For each interception of the other two calls, we review a data structure we maintain to determine if the system call is dealing with an already opened input device. This information can then indicate exactly how much input was requested by processes and subsequently delivered by the kernel.

4.2 Implementation

To capture the applications which aim to capture input from an end-user, we identified the two paths created by the kernel through which processes can access the input devices. These paths, `/dev/input` and `/dev/tty`, are part of the peusdo-filesystem maintained by the kernel for providing processes with a way to interact with the hardware abstraction powered by the kernel. In order to actually obtain data from these paths though, the process must invoke the standard file access system calls. As such, we are particularly interested in the `open`, `read`, and `close` system calls when they are called on these paths. When a process performs such an action, it is a clear indicator that the process wishes to receive data from the input devices. In order to log this information, our approach studied two possible routes: using the Linux Auditing Subsystem, or System Call Interception.

Our first possible implementation route was through the Linux Auditing Subsystem. The auditing system is a series of logging functions embedded within the code of the kernel capable of logging a number of events being processed by the kernel: for example, it can log file opens, system calls, user logins, activity on filesystem paths, etc. When developing our kernel instrumentation we investigated the possibility of leveraging the auditing subsystem to record which processes open the input devices for reading, but later came to the conclusion that the capabilities of the auditing platform were not quite what we needed. Though the audit system has most of these features, we needed lightweight instrumentation which would process a particular set of system calls, and of those system calls, only process the calls which were dealing with the input devices. For these reasons we decided to create a kernel module.

Our second implementation route, the creation of a kernel module, works by intercepting a process' invocations of the system calls. Kernel modules are blocks of code injected into the kernel and provide additional functionality to the core of the kernel. The purpose of our kernel module is to replace the `open`, `read`, and `close` system calls and record applications which invoke those calls on the particular file paths relating to the input devices. When our code is injected into the kernel it performs three initialization tasks:

1. Searches for the existing system call definitions residing in the kernel.
2. Disables read-only memory and replaces the definitions for `open`, `read`, and `close` and inserts copies of those definitions pointing to the implementations within our kernel module.
3. Re-enables read only memory and beings accepting invocations of our `open`, `read`, and `close` copies.

Each of the copied system calls within our module are short functions, they simply record the necessary information, and then invoke the kernel's original copy. This allows us to maintain full compatibility with every process running within the kernel, but also perform arbitrary logging. We modified the `/etc/modules` file to ensure that our kernel module is automatically started during the boot process.

4.2.1 Open, Read, Close

To extract the appropriate information from the system call invocations, we discuss each system call in turn and data recorded.

The `open` system call has three parameters: the file path to be opened, if the file should be opened for read or writing, and if a new file- the permissions the file should be created with. The system call returns a unique-to-process "file descriptor" which can be used in future system calls to work with the opened file. We compare the file path parameter to two file paths known for holding input devices: `/dev/tty` and `/dev/input`. If the parameter does not start with either of those two paths we simply ignore the invocation and allow it to proceed as normal. If the path is for an input device, then we record the full file path, the process' unique identifier, the name of the executable, and the returned file descriptor. Further, we maintain a small data structure, which we will refer to as "openInputs", to track which process and file descriptor are associated with an opened input device. We output our logged data to the kernel log using the kernel's `printk` function and let the existing system infrastructure save it to disk.

The read system call has three parameters: the file descriptor (as provided by the return of the `open` syscall), the location in application memory space where the read should save data, and the maximum number of bytes the requesting application wishes to receive. In our approach, we only process the syscall if there is an existing entry in our `openInputs` data structure which matches the process and file descriptor provided in the read system call's parameters. If there is a match, we log the requesting process' unique identifier, the associated file descriptor, and the number of bytes read. In doing so, we are able to record exactly the number of bytes read by the application from the input devices. As with the read system call, we perform our logging to disk via the kernel's `printk` infrastructure.

The third and final call we intercept is the `close` system call. It accepts just a single input parameter: the file descriptor (as provided by the return of the `open` syscall) to close. Our approach only logs a process' invocation of this system call when there exists and entry in our `openInputs` data structure which match the requesting process' unique identifier and associated file descriptor. When this happens, we simply log that the application has ended its read of the input device and we remove the appropriate entry from our `openInputs` structure.

4.3 Evaluation

In order to evaluate our implementation of the kernel sensor, we inspect both the performance overhead of the approach, and the efficacy of the logging. We perform a test with manual use of the keyboard and mouse, and an inspection of the results to verify the expected processes are captured.

4.3.1 Performance

Since our instrumentation is operating within the kernel, a heavily optimized and time-critical portion of the system, our approach needs to log data quickly. This becomes especially important as the particular system calls our module intercepts are very frequently used during the use of an end-host. We performed two tests on a Ubuntu Desktop 14.04 virtual machine with four 2.6GHz cores and 2GB of RAM to verify our performance is within acceptable bounds. First, we test how long it takes for our instrumentation to execute when a process opens, reads, or closes a file path dealing with an input device. We show these results in Table 1. Secondly, we test our performance cost when we intercept a system call invocation which does not deal with an input device and

Table 1: Overhead time in μs when processing an `open`, `read`, or `close` system call handling input from user input-devices as seen through 28,265 data points. 18 zero entries, and 105 large data points excluded.

Minimum	Median	Average	95th Percentile	Maximum	Standard Deviation
0.024	0.643	0.680	1.317	3.983	0.347

Table 2: Overhead time in μs to determine an invocation of an `open`, `read`, or `close` system call is *not* dealing with user input as seen through 42,872,901 data points. We excluded 304,236 (0.7% of total) '0' entries and 15,875 entries for being greater than two times the median.

Minimum	1st Percentile	Median	Average	99th Percentile	Maximum	Std. Dev.
0.001	0.231	0.387	0.377	1.052	4.949	0.193

therefore does not require further processing. These results can be found in Table 2. To capture the real, or wall-clock time, our code takes to complete, we use the kernel's `getnstimeofday` before and after our code executes, then log the calculated difference.

When displaying our performance results we remove data points which claimed to take no time ("0"), and data points which were over two times the standard deviation greater than the mean. We attribute the results which indicated a "0" overhead due to faults, or insufficient resolution, in the kernel's `getnstimeofday` as a "0" would indicate the unlikely result that the kernel completed the entire task within three clock cycles. We attribute the large and comparatively very slow results, to operating system scheduling quirks (e.g. another process executed which paused our code) rather than delays in our kernel module.

We can see in Tables 1 and 2 that our approach has extremely small overhead when handling events passing through our kernel module. When either deciding that a system call needs to be logged or determining that it does not we are able to pass control back to the normal kernel process within, on average, a few hundred nanoseconds. Even with system calls being invoked frequently, our logging of the appropriate information to determine which processes are reading from the input devices causes imperceptible delays for the user.

4.3.2 Efficacy of Monitoring

To determine if our method of instrumentation and selective data logging were effective in identifying the processes which read input devices, we performed two verification tests. Our first test shows we are properly recording incoming input proportional to the input created. Our second test verified that we were properly detecting applications which were reading from the input devices.

We ran our first test by powering up a Linux Ubuntu Desktop 14.04 instance, letting it boot to the login screen, and performing two simple sets of actions. First, we simply moved the mouse around for an increasing amount of time. Second, we type arbitrary keys on the keyboard at the rate of one key press every second. We capture results every twenty seconds over the course of a minute and a half. We represent our results tabularly in Table 3.

Table 3: Results of moving the mouse around and pressing a single keyboard key every second. Results taken every twenty seconds over the course of nearly two minutes. Our kernel sensor accurately reports the increase in bytes read over time.

	Number of Bytes Read from Device				
Device	Start	20 Seconds	40 Seconds	60 seconds	80 seconds
Mouse	23	1734	3951	6388	8631
Keyboard	29	67	107	146	186

Our results indicate that we not only successfully capture the input data read from the input devices, we also record results proportional to the amount of data received even across multiple `read` system calls. In Table 3 we notice an approximately linear increase for the capture of mouse movements for eighty seconds during which time we were constantly moving the mouse around the screen. Despite this trend, we notice less events when moving the cursor slowly compared to moving the cursor at a rapid pace. This leads us to believe that the mouse reports movements are mainly reported at a set frequency, though the frequency can increase for rapid movement. We note that all testing was done with an virtualized absolute pointing device powered by `qemu` [33], and other mice might report movement differently. Similar to the mouse, our results also show an approximately linear increase for the capture of keyboard events. As opposed to the mouse however, a linear trend is expected for the keyboard. This due to how the keyboard state is maintained: either a key is pressed, or it is not. By pressing a single key approximately every second we record the information regarding the change of state to reflect the press of the key, then another change when the key is release. Accordingly, we notice two events for each keystroke in a linear trend following the number of keystrokes.

After rebooting the Ubuntu instance, we performed our second test focused on the effectiveness of our ability to capture which processes read from the input devices. We waited for the Ubuntu display manager (the login window) to start, then over an SSH connection we ran the command ‘`sudo cat /dev/device/mice > /dev/null`’. This command will open the mouse device, read all input from the device, then discard the output (bytes written to `/dev/null` simply get dropped by the kernel). Under these conditions we expect to see two applications receiving input from the mouse: the X.Org Window System (powering the login window), and our own `cat` process. We proceed to move the mouse around and click a few times before finishing the test. We then repeated the process but used the following command to read input from the keyboard instead of the mouse: ‘`sudo cat /dev/device/input0 > /dev/null`’. We pressed and released a number of keys on the keyboard, then ended the test. We inspected our log file, noted the programs which opened the input devices and calculated the sum of bytes each program read.

We first inspect which programs opened the input devices. Interestingly we observed a number of programs opening and reading from `/dev/tty`. According to the Linux documentation, the `/dev/tty` alternative tty device is the ‘current’ tty associated with the process running [34]. As these are not connected to the user-input devices, we ignore the entries in our logs referring to `/dev/tty` device. In in the abbreviated log entry below, we can see how our kernel sensor logged the input devices being opened by processes:

```
pid:3922 fd:3 exe:sh file:/dev/tty7
pid:3961 fd:7 exe:X file:/dev/tty7
pid:3961 fd:10 exe:X file:/dev/input/mice
pid:3961 fd:10 exe:X file:/dev/input/mice
pid:3961 fd:10 exe:X file:/dev/input/mice
pid:3956 fd:3 exe:sh file:/dev/tty7
pid:4971 fd:3 exe:cat file:/dev/input/mice
pid:4976 fd:3 exe:cat file:/dev/tty7
```

We then are able to inspect our logs for the amount of data each of these devices read. Below is an except from our kernel log as an example for how we record process' reads:

```
read pid:4971 fd:3 bytes:3
read pid:3961 fd:10 bytes:4
[ ... ]
pid:3961 fd:7 bytes:2
[ ... ]
pid:4976 fd:3 bytes:2
```

As can be seen in Table 4, not every application which opens our input devices actually read input. We can see this in the two cases of the shell program `sh` opening, but never reading, the input devices. On the other hand, we see that the program `X` read both from the keyboard and the mouse. We also see that out experimental reads using the program `cat` appear in our results as well. As an interesting side effect of our tests, we can see that our `cat` program received more data than the program `X`. This a functionality of the `/dev/tty7` device. It only allows for a single program to read input from it at a time; before our test, `X` was receiving keyboard input, during our test only our `cat` program received input, and at the end of our test `X` began receiving all the input again.

Through the use of our kernel sensor we are able to extract and log which applications read from the input devices. However, the data mainly indicates that there is a single large consumer of keyboard and mouse data, the program `X`. On Linux systems, instead of each program receiving user-input directly from the kernel, all the input data is sent to the X.Org's X Window System [7]. As the X.Org X Window System is responsible for creating the graphical user interface and managing which processes appear on the screen, it consumes all of the user input and dispatches it to the correct recipient makes sense. To detect which processes actually receive the user input, we place instrumentation within the X Window System.

5 Second Sensor: X11

The second sensor of our approach is embedded in the code of X.Org's X Window Server (X11), a widely used windowing system for Linux platforms [7]. On mainstream Linux distributions X11 is installed and tasked with handling the display of windows, managing the mouse and keyboard,

Table 4: Input-device data read by each process as recorded by the kernel sensor.

Program	PID	Mouse Reads	Keyboard Reads	Total Reads
sh	3922	0	0	0
X	3961	2464	86	2550
sh	3956	0	0	0
cat	4971	2170	0	2170
cat	4976	0	120	120

and working across many types of hardware. X11 is particularly useful for our approach because it is in the middle of the interaction between an end-user and the applications displayed on the screen. Additionally, since X11 is an open-source project, our approach is able to freely review and instrument the appropriate code to make our sensor work.

On our chosen implementation platform, Ubuntu Desktop, X11 is owned and executed under root permissions. As such, we place our X11 instrumentation within our trusted computing base when we exclude root-level compromises. Despite this, our instrumentation does provide some quantitative metrics (keyboard presses, mouse movements) which can be correlated with our kernel sensor to enhance trust. Additionally, while our approach instruments X11 in a way which allows the partial logging of remotely connected applications, we assume that all clients of the X windowing server are applications running on localhost (e.g., we exclude X11 session tunneling).

Through our X11 sensor we are able to extract the processes that are connected to the X11 Server, and of those processes, which ones received input from the user. We do so by tracking which on-screen windows each client application creates and then logging which window receives an input event. Furthermore, we are able to detect if a process' window is fully viewable on the screen, partially viewable, or completely obscured when receiving an event. Associating which window, fully visible or not, receives user input is critical for the accurate attribution of which process received input. For example, a user can scroll up, down, and trigger mouse-over events on background windows. Our sensor also keeps track of how frequently an application sends an update to its on-screen windows: by using this information we are able to quantify the amount of output a process shows to the user. In combining both of our quantitative metrics in describing both the user input and process output, we can classify progress as either "input-heavy," "output-heavy," or "balanced."

5.1 Approach

Through the kernel sensor we are able to detect which processes open and read from the input devices, but in default deployments the X Window Server is the only consumer of such input. In order to continue following the chain from the user to the process, we instrument X11 to indicate which processes are registered with, and listening for events from, X11. We now discuss the current state of the graphics processing chain on our chosen environment, describe the purpose of X11, and how our sensor works.

The kernel provides a layer of abstraction between the display hardware and processes wishing to utilize the display. In Linux, a framebuffer is the layer of abstraction for the screen. As with any device abstracted by the Linux kernel, the framebuffer can be thought of as a file located at `/dev/fb0` and interacted with using the standard file system calls `read`, and `write`. In essence, it is a region in memory where an application can write the color values of individual pixels and then expect the kernel to perform the appropriate manipulations for the data to appear on the screen in front of the user. It is worth noting though that the use of the kernel framebuffer is not required; in fact, newer graphics hardware will expose their own framebuffer equivalent for applications to interact with.

5.1.1 The X Server

While the framebuffer is nice abstraction by the kernel, the framebuffer gives any single application access to the entirety of a display. This can be useful in cases when a process (or the kernel) wishes to display "full-screen" applications (e.g., the system-owned virtual terminals). However, this becomes problematic if you want to run multiple applications on the same screen. If two processes open the framebuffer and attempt to write their own application window, corrupted data will be printed to the screen as each process fights for the same space. To solve this issue a third process is required. This third process must be capable of receiving graphical commands from each application and multiplexing them correctly onto the framebuffer. This lead to the development of X11.

The X Window Server created by the developers of X.Org is an open-source implementation of the X Window Server specification. Commonly referred to as X11, it creates layer of abstraction between the framebuffer provided by the Linux kernel and applications wishing to have a screen presence. The X server simply creates a screen of resolution equal to that of the kernel framebuffer and accepts requests for drawing to the screen from client processes. At a basic level, to render application windows on screen, applications must interact with the X server or risk colliding on writes to the framebuffer. Furthermore, any application wishing to write to the framebuffer must have root-level permissions.

In order to facilitate any number of possible applications requiring the screen to be multiplexed, X11 works under the Client-Server model. Th "XServer" component controls some number of displays on the host in which it is running. The server is able to accept connections from clients who implement the X11 protocol; typically (for security reasons) these connections are only allowed from localhost over unix networking sockets. Any number of clients can connect to the XServer using an Application Programming Interface (API) exposed by X11. This API, commonly referred to as "Xlib," contains the information and methods required to speak the X protocol with XServers. Furthermore, this API contains abstractions such as a 'window' to simplify a process' requirements to draw to the screen instead of requiring them to manage the raw pixel information that would accompany writing directly to the framebuffer. Applications wishing to display something on the screen may use Xlib to create a connection to the XServer. Once the connection is formed we refer to the client process as an XClient.

5.1.2 Window Managers

X11 was designed under the mentality that a Windowing System should do no more than is strictly necessary to provide just the windowing features. This lead to the design of X being somewhat minimalistic, but with the support for an empowered XClient to dictate how it wanted all windows to appear. For example, there is the standard desktop-emulating windowing approach where individual windows can be overlapped partially or even fully, just like you could with papers on a desk. Alternatively, X11 can also power a windowing approach where all windows are snapped to specific locations and sizes without any overlap. These alternative windowing approaches are preferred by some users for a more efficient work environment. As an example, Microsoft Windows 7 has a sense of this with it's snap-window-to-edge feature "Snap" [35].

In order to support this wide range of windowing mentalities, X allows for a single XClient to connect and request special window management features. These special permissions allow for certain types of control over all windows being displayed. XClients which invoke these permissions are referred to as Window Managers. Window Managers come in many different forms, but for our approach with Ubuntu Linux, `compiz` is the default installed window manager [36]. On Ubuntu, `compiz` is responsible for adding the standardized window decoration and provides control over the relocation of windows across the display (typically by drag-dropping the title bar of the window). The Window Managers are also able to reparent the root window, meaning they can place their own window as the top-most window on the desktop and influence all other windows on the screen. However, the main dispatching and handling of client actions are still routed through the core of X11 and thus our approach does need not need to instrument the window manager as well.

5.1.3 Display Managers and Launching X

In addition to the Window Manager, most distributions will also come packaged with a Display Manager. Launched right after the kernel boot process finishes, the Display Manager is responsible for displaying a graphical user interface (GUI) for the end-user to authenticate with (typically referred to as the login screen). This GUI replaces the standard text-based console login that would otherwise be present. The display manager starts an XServer instance, and communicates with it in order to display the login greeter and authenticate the user over the special X Display Manager Control Protocol [37]. Once a user is authenticated, the display manager will start a new XServer instance to handle the display of the user's desktop and applications the user wishes to run. On our chosen implementation platform, Ubuntu 14.04, the default display manager is `lightdm` [38].

5.1.4 Placing Windows on the Screen

An application must go through a number of steps in order to display a window on the screen through the XServer. To aid in understanding how the system works, we will discuss the method used by an application to display a window on the screen and how such a process factors into our instrumentation approach. First an application includes the Xlib header file, which provides all the functions to interact with the XServer. The application then opens a connection to the display and prepares to create a drawable object. In the world of the X Window System, a drawable is one of two objects: either a pixmap, or a window. Both are objects which can receive any of the commands which create a displayable visual; for example, the application can instruct the XServer to create a line on a specified drawable.

While both drawable objects, the main difference between a pixmap and a window is that only a window is actually displayable on the screen. Typically, an application will create a pixmap for off-screen generation of what it wishes to be displayed, then instruct the XServer to simply copy the completed contents of a pixmap over to a window to be displayed on the screen. This provides a number of benefits to the application; for example the application can hide any intermediary alterations to the window and only provide a completed update when it is available. This use of the pixmap is a common approach to reduce flickering or tearing in user-visible windows and often referred to as ‘double-buffering.’

With a newly allocated window, the process next needs to listen for events on the newly allocated window from the XServer. Usually this is done by the client requesting notifications for only the events it cares about. These include an “Expose” event in which the process is told that the window has become at least partially exposed to the user. When an XClient, and by extension the window, registers to accept an Expose event, it can react to such an event by telling the XServer to, for example, copy a portion of it’s off-screen pixmap into the now visible part of the on-screen window. Finally, to place the allocated window on the display, the process indicates to the XServer that the window should be mapped to a particular display. Once this final step is performed the window will be viewable by the user.

In order to track and quantitatively reason about a process’ screen presence, our approach instruments the key portions of this process. In particular, we focus on the XServer’s processes for handling the creations of windows and copying of drawables.

5.1.5 Listening for User Input

Similar to listening for Expose events, applications can also indicate to the XServer that they wish to be notified of user input events. The X Window Server supports a wide range of input types, ranging from the standard keyboard and mouse all the way to touch screens and tablets. For the purposes of our approach, we restrict our focus simply to the keyboard and mouse. In order to request input notifications, the client application must set an event mask on windows it has access to. This informs the XServer that on that particular window the process has requested notification of user input. This means a process which has registered to receive input from the keyboard and mouse will only be passed the information if the particular keyboard press or mouse movement occurs within the specific window the process has registered on.

Since an application can only receive input events on a window it has registered on, we are able to track which applications receive input from the user by watching the flow of events. For instance, if we see that a particular XClient has received thirty-four keyboard press notifications, we know that the process associated with that client connection has been notified of user input. On a larger scale, by logging all delivered events relating to user input, we are able to indicate which processes received which types of events, how frequently, and indicate the quantity of events each process received.

To perform the instrumentation necessary to both quantitatively record an application's display output and user input, we target the XServer code responsible for handling the dispatching of application requests, and the delivery of events.

5.2 Implementation

We explored the ability to instrument the X11 source code on the Ubuntu Linux distribution, release 14.04. We chose this particular release of Ubuntu because it is on Ubuntu's long term support schedule. This means that the engineering we perform in determining the viability of our X11 instrumentation will be usable for the duration of the support cycle (approximately five years) [39]. Ubuntu's 14.04 release comes packaged with a slightly altered X Windows Server based off of version 1.15.1 as minor alterations are performed by Ubuntu developers to backport security updates. In an attempt to use the same version packaged by Ubuntu, we downloaded the source code which the maintainers of Ubuntu use when compiling the binary package shipped with the release. However due to compilation errors and other miscellaneous issues, we instead opted to download the latest source straight from the maintainers of the X.Org X Windowing System, the X.Org foundation [7].

Our checkout of the latest X Windowing Server source code provided us with a stable, but in-progress version based on the 1.17.1 release. To perform the automated checkout and build process, we employed a shell script provided by the Xorg Foundation [40]. While our version is two releases ahead of the version packaged with the Ubuntu 14.04 distribution, there were no issues compiling and running the software. After building the X Window Server portion of the project, we installed the new binaries to a custom prefix to leave the original Ubuntu-packaged installation untouched. We altered the `lightdm` configuration file, `/usr/share/lightdm/lightdm.conf.d/50-xserver-command.conf`, to boot our custom compiled XServer instead of the original one. With this setup we were able to rapidly prototype our alterations to the code and test them on a virtual machine.

5.2.1 Capturing Graphical Output

In order to track and quantitatively describe the graphical output of processes, we reviewed the XServer codebase. As all XClients need to communicate the desired window alterations to the XServer, we focused on instrumenting the code handling this client-server conversation. Within the `xserver/dix/dispatch.c` source file we located and placed logging hooks within the function appropriately named `Dispatch`. This XServer function is responsible for looping over all open file descriptors and appropriately handling any incoming data. As requests come in from clients, they are dispatched accordingly using a table of functions pointers. The table of function pointers holds the mapping between the X protocol's request numbers and the correct function to be executed.

We identified a number of request handling functions potentially useful for our approach. The list is as follows: CreateWindow, CopyArea, DestoryWindow, PolyFillArc, PolyFillRectangle, PolyLine, PolyPoint, PolyRectangle, PolySegment, PolyText, PutImage. After some inspection, we decided that it was not needed to embed instrumentation in all of those functions. Instead, we identified the following three functions as frequently used and good enough to provide the data we needed:

- CreateWindow: Create a drawable Window of the specified dimensions and properties.
- CopyArea: Copies the contents of one drawable to another.
- PolyLine: Creates a single line from point 'a' to point 'b'

We chose these particular requests because they covered a reasonable subset of the possible options and would allow us to determine whether or not the thematically-related requests were used at all. We determined after a quick informal test that, as expected, CreateWindow is used frequently. There were many calls to CopyArea moving display data from pixmaps into windows and there were no requests for PolyLine. We attribute the high use of CopyArea to the use of user-space graphical libraries which take care of constructing and rendering the graphics to a pixmap on the XServer. When the application is ready, it sends a request to the XServer indicating the drawn frame should be copied from the pixmap to some on-screen window.

For those windows displayed on the screen, there will be some windows which are actually visible, some windows which are covered, and others which are completely covered by other windows. In our approach, we seek to separate these fully visible windows from those which are not. Even if a process has some screen presence, if the screen presence is always covered by some other window, then we want to be able to represent that. To do so, we leverage the window attributes maintained by the XServer. For every window allocated by a client, the XServer creates a new `_Window` struct. This struct contains everything the server needs to know about a window. For our purposes we are particularly interested in the visibility, drawable-width, and drawable-height fields.

By processing these three attributes about a window, our approach is empowered with the ability to differentiate between processes which actually have a screen presence and those which simply have sent a few commands to the XServer. The drawable-height and drawable-width fields record, as an integer, the height and width of the window in pixels. The visibility field consists of four possible values representing the current visibility state of the window.

1. The window is completely *un*obscured i.e. the window is fully shown to the user.
2. The window is partially obscured, perhaps just by a few pixels, or perhaps nearly all covered. We leave the computation of exactly "how much" of a window is visible to future work. In our approach we simply note if a window was visible at all.
3. The window is totally obscured and not visible to the user.

4. The window is not viewable.

This last state is usually caused by a client creating a window but not completing the process to "map" and cause a window to be a candidate for display on the screen.

When our X11 sensor is triggered from the three functions in which we embedded instrumentation, we record the current state of the window in question, the window's width and height, the window's unique identifier, and the unique identifier for the client connection. Through the aggregation of these log entires, we are able to determine which processes sent data to the screen for display and which processes did not.

5.2.2 Capturing User Input

On the reverse path, tracking user input to processes, our approach focused on a different source file in the XServer code. As discussed previously, we know that processes are required to "register" or set event masks in the windows they manage if they wish to accept events, such as user-input, on those windows. Our approach was to track backward from the code within the XServer which delivered the events to the window's clients up to the point where it was computed which events were desired by which clients. In other words, we located the portion of XServer code which compared all generated events on a window to the list of event-masks and determined which events were dropped (i.e. no one wished to receive the event) or which events were to be transmitted down to the window's clients. We note while the XServer code does indicate a window can have multiple clients, we never saw this practice occur; for simplicity we assume that a window has only a single client.

If an event is determined to be desired by a process, we aim to capture and log the events shown in the list below. For some logically-equivalent event types, there are multiple versions of the event (e.g., `DeviceKeyPress` and `KeyPress`). While both these events convey the same information, they have different sources in the XServer code and therefore are named separately. In the list below we show these events with a similar meaning as a single group.

- **DeviceKeyPress, KeyPress**: This event occurs when a keyboard indicates that a key has been pressed down. We log the which key on the keyboard was pressed.
- **DeviceKeyRelease, KeyRelease**: This event occurs when a keyboard indicates that a previously pressed key has been released. We log which key on the keyboard was released.
- **ButtonPress**: This event occurs when the mouse indicates that a button has been pressed. The button could be the left, middle, or right mouse buttons.
- **ButtonRelease**: This event occurs when the mouse indicates that a previously pressed button has been released.
- **MotionNotify**: This event occurs when the mouse indicates motion.
- **GenericEvent**: This event occurs when the event was generated by an X11 extension.

Using these events generated and delivered by the XServer, we are able to quantitatively indicate the amount of user input a particular window receives. By extension, we obtain the amount of input a process receives. In particular, we log two events for each keyboard and mouse button inputs as both KeyPress and ButtonPress events have their corresponding KeyRelease and ButtonRelease events. Further, the MotionNotify event is triggered frequently and appears to be bounded by time intervals, not by the rate or distance the mouse traveled. We log each event and determine amount of mouse movement within a window by the number of MotionNotify events the window receives.

We track the GenericEvent type due to a particular X11 extension which is sometimes involved in handling the user input. The `Xi` extension is an attempt to increase the types of input devices the XServer can handle [41]. For example, it provides support for the keyboard, mouse, trackpad, touchscreen, tablet, and even multitouch devices. While we assume just the standard keyboard and mouse are present, our approach made a concerted effort to capture the events from this extension because it can occasionally be used to deliver keyboard and mouse events under certain process conditions, such as when the user is at the login screen. In our implementation, we are able to simply recognize the `Xi` input identifier, and then apply our existing code to determine if the contained event is a KeyPress, KeyRelease, etc.

In all, we log the event type, the unique window identifier, whether or not the window is visible, and the file descriptor for the client connection receive the input. If the event is for a KeyPress or KeyRelease, we also log the particular key-code (name of the key) which was pressed/release on the keyboard. Future approaches could leverage this data to track the movement of sensitive keywords or even passwords and identify when such data traveled to programs which should not have received it.

5.2.3 Correlating PIDs with Windows

A major portion of our instrumentation relies on the ability to take a `_Client` struct, representing an XClient to the XServer and output the associated process. Determining which process corresponds to which XClient is not a simple task and requires some extra engineering. The X11 client-server protocol was written under the assumption that the XServer and XClient may not be on the same physical host, and as such, maintaining state on a processes' unique-to-system identifier (PID) holds no meaning. For example, there is no way to know the PID of the process if the XClient is on a remote machine. However, since our approach assumes all XServers and XClients are hosted on the same physical machine, knowing the PID of the XClient is not only useful, it is vital to knowing which process is interacting with the user.

In a general move towards tracking the process identifier of connected XClients, X11 developers added in a new window property: `_NET_WM_PID`. This property is a newer addition to the X Window System and works in conjunction with `WM_CLIENT_MACHINE`. The general idea is that if `WM_CLIENT_MACHINE` is set to the host's own hostname, then the `_NET_VM_PID` is the pid of the process running on the host. If the hostname does not match the host's own name, then the PID is representing the process identifier on some remote host, and is not helpful. Since in our approach we assume all XClients and XServers are on the same host, we can always get useful meaning out of the _NET_VM_PID property. However, the property must be set by the XClient process itself. If the process either does not support it, or decides to give a false PID value, the XServer will not be able to tell the true value. Therefore our approach needed a different method to obtain the true PID of the XClient.

To reliably detect which process is responsible for initiating an XClient session, we focus on the mechanism of communication. When on the same host, the connection between the XServer and XClient occur over UNIX sockets. Just like normal Internet-style sockets, the UNIX sockets are simply a connection between two processes. Our approach is to retrieve the details of this connection from the XServer and follow the connection back to the originating process. In order to do so, we retrieve the file descriptor for the XServer's side of the socket from the `_Client` data structure.

Gathering information regarding sockets and their connections requires assistance from the kernel. To gain this information, we first retrieve the system-wide unique identifier for the UNIX socket associated with the particular XServer-XClient communication channel we have targeted. To do so, we simply perform a `read_link` system call on the file path `/proc/self/fd/##`. This utilizes the kernel's pseudo-filesystem "proc", to tell us information about the current process "self" (X11), with respect to the file descriptor "##" (obtained from the `_Client` struct). When reading this pseudo-link we are returned the unique UNIX socket identifier. With this in hand, we can then use the existing socket tool `ss` to query the kernel and extract the connection information relating to the specific UNIX socket in hand [42]. With this data returned, our instrumentation will have reversed the socket connection and learned of the true process responsible for the client connection.

Unfortunately, performing the computation to connect a client with its process takes enough time to introduce noticeable slow downs in the user interface. To overcome this, we place instrumentation in the key location to perform the computation a single time: immediately after the XServer establishes the client connection. By placing the instrumentation within that block code we can be sure that there will not be an event or request handled by the XServer without first ensuring we know which process is connected as the XClient. Furthermore, we are able to detect when a client disconnects from the server and appropriately log the recycling of file descriptor to retain our log integrity.

5.2.4 Creation of the Logging Infrastructure

To augment the instrumentation of the XServer, we added a logging mechanism to the code of the XServer. Our mechanism was designed to take all of the logging data we create and save it to a single file on the disk for later processing. We designed our logging functions to be similar to the `printf` function in that our functions are able to consume an arbitrary number of typed parameters. After some processing of the logged data, such as adding a timestamp and performance data, we leverage the existing `vfprintf` function to save the logged data to an open file descriptor.

We implemented full pthread-based mutexes to protect our log file from race conditions which could corrupt our saved data.

5.3 Results and Evaluation

We evaluate the effectiveness of our X11 sensor through two methods. First, we review the performance cost of our instrumentation and discuss the effects, if any, such overhead has on the user's experience. Second, we determine how well the sensor is able to extract meaningful interaction data from the Xserver. In particular, we show our instrumentation is sufficient to tell the difference between processes with no user interaction and processes which had some interaction.

5.3.1 Performance

Since our X11 sensor is in the cornerstone of the user interface, our performance is critical to the overall speed of the GUI. This can become problematic if, for example, our sensor causes any undue delays while extracting the interaction information from the data flowing through the XServer as it will be directly observable by the user. To aid in expediting our processing we leveraged a few mechanisms for efficiency: stateless data extraction, buffered data logging to disk, and minimizing duplicated work. In doing so, we push the majority of the analytic computation to either a separate process reading our log in real-time, or to post-processing work. To measure our overhead when extracting data from the XServer, we include a performance metric within each entry of information we log. Utilizing programming libraries included with Ubuntu we invoke `gettimeofday` to obtain a microsecond resolution timestamp immediately when our data extraction begins, and then again immediately after the extraction has completed. We compute the difference in 'wall-clock' time and then include this difference at the end of our log entry as it is buffered for output to disk.

We collected performance results by running a test in which we exercised a number of applications while using our instrumented version of the XServer. After collecting the performance data, we first removed 73 entries within our performance logs which indicated that our instrumentation took no time; these '0' entries we attribute to the system time libraries not having sufficient resolution to accurate measure the elapsed time. Second, we removed any entries which were determined to be over two times the standard deviation greater than the mean. These numbers we attribute to operating system scheduling factors causing our instrumentation to wait while other processes completed. In all, we only removed 2,515 entries, or 0.17% of the data points. After the removal of these outliers a summary of our performance results can be seen in Table 5. Through our performance analysis we conclude that, on average, our X11 instrumentation takes approximately 20 microseconds to complete. We believe such a low performance overhead will cause no delays or other issues with rapidly processing user commands to update the interface.

Table 5: Overhead time for each logged user-event and process-request in μs as seen through 1,516,861 data points. 73 zero entires and 2,442 entries excluded from the statistics for being two times the standard deviation greater than the original mean.

Minimum	5th Percentile	Median	Average	95th Percentile	Maximum	Std. Dev.
1	3	7	19.46	33	988	57.16

5.3.2 Efficacy

We next tested how well our X11 sensor was able to detect the processes with which the user interacted. In order to test this, we conducted a controlled experiment where we opened a set of applications, used them for their intended purpose, then closed them. Based upon the intended purpose, we would expect to see certain behaviors. For example, the text-editor `gedit` we expect to accept a large amount of input from the user while typing or revising a document. On the other hand, we would expect a video player such as `vlc` to have a high level of output to display a video and little user-input as a user need only click play. By analyzing our X11 sensor's log data we obtained the results below; we indicate the name of the client process, the process' unique identifier, the quantity of display output, and the quantity of input delivered (separated by input from the keyboard and input from the mouse). We abbreviate the list by specifying applications with no more than 20 output events, and no input events should be ignored.

```
Client              PID       Output     Keyboard      Mouse
"chromium-browse"   12703     5769       0             1382
"vlc"               12127     231        0             0
"vlc"               12127     1034       0             0
"vlc"               12127     5780       0             0
"gedit"             12060     3892       4031          137
"nautilus"          11520     1050       4             477
"compiz"            11472     42568      0             1
"bamfdaemon"        10746     2          99            156
"unity-greeter"     10042     64         54            4
```

In the initial our of testing our prototype we began to see trends forming in the relationships between categories of applications and the amount of user interface input/output they performed. As with any such behavior-based categorization, our classification of processes into groups of interaction types is heavily influenced by the user and their particular usage patterns. For example, editing a document in the Google Documents suite will make `chromium-browser` appear with relatively more input events compared to watching a series of YouTube videos. Despite this possibility, we were able to take advantage of repeating trends to create the following classification. We first group "no-interaction" processes by simply listing the processes running on a host and identifying those processes not on the above list. Second, we group programs which are "output-heavy" as processes which have a more than four times the number output events than input events. Third, we group processes which are "balanced" as processes which have between one-third and one-half the amount of input events as output events. Finally, we group processes which are "input-heavy" as those which have at least half of the number of input events as output events.

Based upon our categorization we can reason about the results we received from the X11 sensor. For example, we can make the statement that in our trial `chromium-browser` is considered an output-heavy application with 24% of its output events being matched with input events, but that it is close to being balanced. This makes sense given the general purpose of the web-browser is to accept some amount of user-input, then display web-pages, videos, and images back to the user. On the other hand we observe the `gedit` program, which received more input events than outputs events, making it a clear input-heavy process. We can further extend our categorization to processes which are built for handling the graphical interface itself. For example, `unity-greeter` is a process for handling the display of the login-window to users. Intuitively, we reason that the login-window is at least a "balanced" process given that it must display items on screen and accept authentication credentials from the user. However, the login-window is relatively output static and mainly just accepts many input characters (i.e., a password). This matches with the "input-heavy" result we observe in our test results.

Through testing both the performance and efficacy of the X11 sensor, we have shown our implementation to be effective. Our extraction of data is efficient and quick, preventing the user from even recognizing that the logging is taking place. We further show that our sensor is not only able to accurately record user-interactions which processes, it also provides data allowing further reasoning about how a user interacts with the process.

6 Third Sensor: GTK

With the kernel and X11 sensors, we are able to quantitatively track input stemming from the input devices and also know to which process the events are delivered. However, these sensors have no visibility into the process itself nor an understanding of what the user is seeing. As X11 mainly deals with rendered images, it is of no use to understand the context which caused the user to enter an input, or the output which the application generated for the user to see. To gain this insight, we look to the graphics libraries. Most contemporary applications are built upon graphical libraries which are able to handle the details of generating objects, windows, and handling events when creating a graphical user interface. Of the many libraries available, the most commonly used one on the Ubuntu Linux distribution is the GIMP Toolkit, or GTK+ [8].

We focus on GTK+ and its ability to generate interfaces on the behalf of processes. When a process utilizes GTK+'s public API to, for example, create a new window, the process can also include information like the window's title or the window's parent. We leverage this communication between a process and GTK+ to capture text and relationship assignments. By placing instrumentation within the appropriate GTK+ API code we are able to extract contextual information from an application by studying the object relationships and the text sent for display. With this data in hand we are able to start reasoning about the actions a user performed within the graphical interface on an application and compare that to observed system actions.

In this section we review the GTK+ library and discuss our approach for instrumenting all of the appropriate API calls to capture text and hierarchy information from processes. We show that we are able to do so in an efficient way without any observable delays by the user.

6.1 Approach

Graphical user interfaces (GUIs) are designed specifically to be understood by the end user. Applications strive to create a graphical layout which will be intuitive, powerful, and used to quickly complete tasks. We aim to leverage this design philosophy towards systematically extracting information from displayed content and provide the data in a form possible for automated reasoning. While the GUI usually employs many pictures and graphics to convey information, the interface is still highly focused on text. Even applications which attempt to create an interface with a minimal amount of visible text usually have tooltips or other explanations associated with their icons to aid in a user's understanding. In addition, applications typically employ common text phrases or keywords to describe certain actions. In our approach, we take advantage of the GUI's user-focused design. We extract text and hierarchical information to provide context towards understand the relationships between on-screen objects.

In Figure 2, we provide an example dialog modeled off of a common web browser's print dialog. When the user clicks on the print button, we can trigger a crawl over the elements on the dialog and gain an understanding of the purpose of the dialog. We can inspect the text on the button itself, review other buttons on the dialog, inspect the text on the dialog, and study how elements on the dialog relate to each other. In doing so, we are able to create rich context describing the graphical layout of the dialog. With rich context in hand, we are able to connect what a user interacted with on the screen with behaviors of the system. In particular, we can compare logs of an application's system calls with the associated context that appeared on the screen. For example, if the context shows a print dialog, we may expect a connection to a network printer.

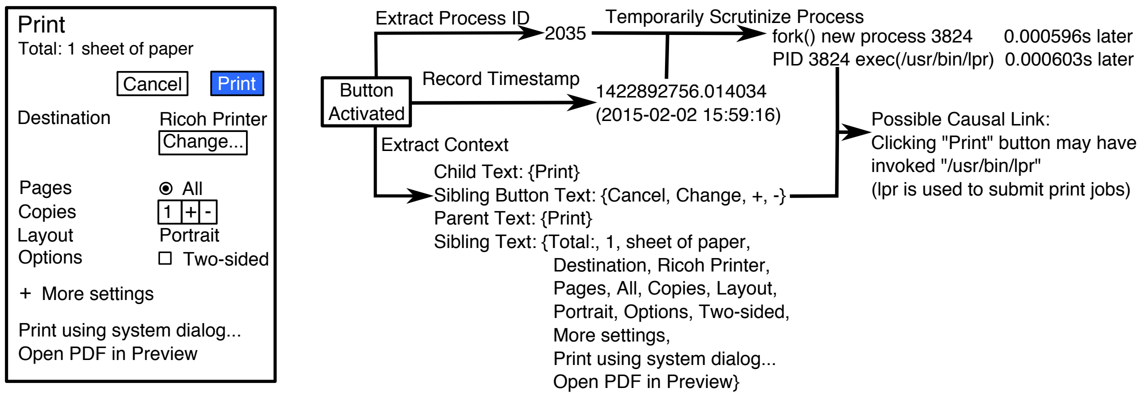
Print
Total: 1 sheet of paper
Cancel
Print
Destination
Ricoh Printer
Change...
Pages
All
Copies
1 + -
Layout
Portrait
Options
Two-sided
+ More settings
Print using system dialog...
Open PDF in Preview
Button Activated
Extract Process ID
2035
Temporarily Scrutinize Process
fork() new process 3824 0.000596s later
PID 3824 exec(/usr/bin/lpr) 0.000603s later
Record Timestamp
1422892756.014034
(2015-02-02 15:59:16)
Extract Context
Child Text: {Print}
Sibling Button Text: {Cancel, Change, +, -}
Parent Text: {Print}
Sibling Text: {Total:, 1, sheet of paper, Destination, Ricoh Printer, Pages, All, Copies, Layout, Portrait, Options, Two-sided, More settings, Print using system dialog... Open PDF in Preview}
Possible Causal Link:
Clicking "Print" button may have invoked "/usr/bin/lpr"
(lpr is used to submit print jobs)

Figure 2: We automatically extract contextual information surrounding user actions, such as button clicks, and fuse it with low-level analysis to inform access control systems.

To capture the text and context information we focus on instrumenting graphical libraries. Building upon previous work [24,25] which attempted to process the rendered screen with optical character recognition, we aim much earlier in the process. These graphic libraries create a layer of abstraction between the gritty details of working with a windowing system (such as X) and the application itself. An application can invoke the public application programming interface (API) of these graphics libraries to quickly and simply create displayable and even user-controllable widgets such as a window, button, or text box. As such, application developers have a strong incentive to utilize these libraries: instead of focusing on the details of building a GUI from scratch, they can leave the heavy lifting to a well prepared library. Our approach focuses on placing instrumentation within key points of these graphical libraries in order to log requests from the application.

When testing the viability of our approach, we focus on the GTK+ graphics library running on Ubuntu Linux. We chose GTK+ as it is one of the most popular graphics libraries available for Linux and has support for Mac OS and Windows distributions as well [8]. While our future work will delve into creating instrumentation for additional libraries, we note that our approach for locating and logging the GTK+ public API invocations can be applied to all libraries, not just GTK+.

Through the careful picking of which API calls we instrument, we can gain insight into the graphics displayed on the screen with a minimal amount of overhead. For example, GTK+ has a number of "set_text"-style functions which are used to assign the text from a C-style array of characters to an on-screen object. Additionally, GTK+ has API calls for setting an object's parent and creating new objects as children of other objects. As opposed to rendering functions, which need to be invoked each time an object is moved, updated, or resized, these functions have a relatively low call rate. The reasoning being, these "set text" and hierarchy calls are only required during the creation of an object and afterwards only used if a text update is necessary. By placing instrumentation within just these API calls, we are able to efficiently record both text and hierarchy information without large overhead costs.

With instrumentation in place and extracted data in hand we are able to recreate the context of the graphics displayed on the screen. By uniquely identifying objects through their recorded memory address and timestamp, we can leverage the child-parent mappings to generate an object-relation graph. Once the graph is created we are then able to use standard graph traversal methods such as breadth-first search to further process the data and discover text and other objects sharing the same parent. Combining these results with the process identification number (PID) we are able to reconstruct which program the user interacted with and the context under which the interaction took place. Leveraging the PID, we are then able to correlate interactions with other sensors and even system calls the application perform to get a more complete picture of what the process was doing.

We utilize the Linux Auditing Subsystem to record the system calls invoked by a particular process. Using the auditing subsystem enables us to efficiently keep track of all actions an application performs requiring greater permissions than is available just in user-space (such as connecting to server or opening a file). The Linux Auditing Subsystem is a kernel feature originally developed for high-security compliance based activity audits [9]. It reports various types of events, including the ability to report the list of systems calls an application uses. While our approach is based upon reconstructing contextual information surrounding an event (like a button press), we will discuss possible approaches to connecting our GUI logging infrastructure to the observed system behaviors. This will foster an understanding of the user-based actions occurring on the system.

6.2 Implementation

To test the viability of our approach of instrumenting graphical libraries, we decided to use Ubuntu Linux 12.10 platform running the default unity desktop environment. On the Ubuntu distribution, there are two versions of GTK+ provided: GTK+2 and GTK+3. To provide maintain compatibility with existing applications and emulate an enterprise deployment we aimed to alter the same versions of the libraries as was packaged with the system. Therefore we sought out and downloaded the source code for GTK+ 2.24.16 and GTK+ 3.6.0 from the GTK website [8]. As a number of long standing programs use the GTK+2 library, which runs an API incompatible with GTK+3, we decided to instrument the basic functions in both GTK+2 and GTK+3 graphical libraries. However, to better support newer programs and fully test our approach, we further modified GTK+3 to capture more text flowing through the newer library. Once the instrumentation was in place, we redirected all dynamically linked applications to use our custom compiled GTK+ libraries instead of the system provided ones. Deployers could alter the `/etc/ld.so.conf` file to make this dynamic linking change persistent across machine reboots.

When creating our instrumentation, we first focused on the GTK+ API functions which were capable of manipulating text in widgets. To do so, we instrumented the sixteen highly used basic functions listed below in both the GTK+2 and GTK+3 libraries.

```
gtk_button_set_label
gtk_cell_view_new_with_text
gtk_clipboard_set_text
gtk_editable_insert_text
gtk_entry_buffer_set_text
gtk_entry_set_text
gtk_label_set_text
gtk_label_set_text_with_mnemonic
gtk_menu_item_new_with_label
gtk_menu_item_set_label
gtk_message_dialog_new
gtk_message_dialog_new_with_markup
gtk_real_menu_item_set_label
gtk_text_buffer_insert
gtk_text_buffer_insert_interactive
gtk_text_buffer_set_text
```

By launching various applications which relied on GTK+2 or GTK+3 we verified text extraction was working as expected in both versions of the GTK+ library. While we observed that a significant portion of on-screen text were recorded just by instrumenting these functions, we wanted to increase our text coverage to all text passing through the library. To do so we developed a thorough and systematic approach to identifying all of the API functions which manage text. We note that while we only performed the extended instrumentation on GTK+3, the systematic approach we developed would work equally well for completing the instrumentation on GTK+2 as well.

We began our systematic approach by reading over the GTK+3 documentation to identify traits which all API functions manipulating text had in common. We noted that these functions would accept either a standard C “char” or GTK typedef “gchar” as a parameter whenever the function would be assigning text to a widget. Leveraging this pattern, we created a regular expression capable of recognizing the text parameter and would automatically insert instrumentation code for copying the text passing through the function call to our logging infrastructure. We executed our regular expression based search over all of the GTK+3 header files with the understanding that application developers would only be able to interact with the functions published in headers which excluded static functions solely in the GTK+ source files from needing instrumentation.

Our search for text manipulation functions within the GTK+ header files returned 642 results. We took this list and manually inspected each function returned, and if we reasoned that the discovered function was indeed handling text and was reachable by a developed (i.e., not an internal function) we uncommented the auto-inserted instrumentation. After this manual process, we concluded with a total of 170 instrumented functions dealing with text. In a similar process, we also identified and instrumented 61 functions dealing with the assignment of object relationships, a few examples of these functions can be seen below.

```
gtk_container_add
gtk_container_add_with_properties
gtk_widget_set_parent
gtk_widget_set_parent_window
gtk_widget_unparent
```

Between our discovery of the text manipulating functions and the relationship-altering functions, we were able to fully instrument GTK+3 to report all text and hierarchy present in interfaces created by client applications. Once completed, we took it one step further and instrumented the single GTK+ function responsible for handling a button-click event in order to know when a user interacted with buttons in the interface.

When an application creates or alters parts of their GTK+ powered interface, our instrumentation activates to record the new state of the interface. The alteration of state could be the updating of text assigned to an object, or a change in how current (or new) objects relate to one another. Through the instrumentation of all the text assignment functions and hierarchy altering API functions, we are confident in our ability to track the state of the application's GUI. To keep persistent records of this information, we write our logged data to disk using the GNU `mkstemp` function which will create a temporary file resident on the storage drive until one of our developed scripts iterates over the folder and aggregates data into a permanent location. Each log entry we save contains a nanosecond resolution timestamp recorded at the time of our instrumentation starting, the GTK+ library function name responsible for causing the log entry, the text or hierarchy data to be logged, and the memory address(es) of the objects being handled. Certain log entries also include the calling application's PID and path to the application's executable on the filesystem.

Having the capability to extract not only the text, but also the hierarchy information along with it leads to powerful contextual information. For example, simply knowing the text on the screen can be useful, but it is even better to know if multiple blocks of text are in the same window or in different windows. Further, it can be beneficial to indicate whether or not the text on the screen is held within a display-only widget or if the recorded text is actually from an editable text box and could have been entered by the user. Our instrumentation is capable of extracting sufficient information from the interface to support making these statements.

6.3 Evaluation and Results

To determine the success of our instrumentation, we evaluate our approach both in terms of performance, and in the efficacy of our text extraction. In terms of performance, we examine the overheads of our instrumentation and discuss the possible impact on the user experience. For determining if our instrumentation is effective in extracting text on the screen, we show results of comparing screenshots of an application with the extraction logs.

6.3.1 Efficacy and Accuracy of Text Extraction

After performing our modifications to the GTK+ libraries, we test the instrumentation by executing a number of GTK+ applications. When selecting applications to test, we aimed to represent a reasonable set which would cover normal day-to-day application use by a user.

- `evolution`, an email client
- `epiphany`, a web browser
- `evince`, a PDF viewer
- `gedit`, a text editor
- `gnome-terminal`, a terminal application
- `gnome-calculator`, a simple calculator
- `pidgin`, an instant messaging client

We spent time interacting with each of these applications and employing each for its intended use-case. For example, we used evolution to read email, gedit to open and edit text files, evince to read a PDF, and pidgin to hold an instant messaging conversation. We ensured that we navigated the menu bars and opened up dialog boxes while in each application to fully expose the text in a number of different windows and layouts.

The results of our application tests can be seen in Table 6. When running the applications `pidgin`, `evolution`, `gedit` and `gnome-calculator`, we were able to extract the majority, if not all, of the text displayed on the screen. Most of the text we extracted from the GTK+ library was an exact match to the information we observed on the screen; however, some of the text had an extra underscore ("_") character present. These underscores, referred to as mnemonics by GTK+, are used by application developers to indicate keyboard accelerators (shortcuts) which GTK+ handles [43]. After we removed these excess underscores, our text extraction reached perfect accuracy.

Table 6: Efficacy and accuracy of text extraction across applications.

	Words Appearing in		String Matches	
Application	Screenshots	Extraction	Exact	After '_' Stripped
`evolution`	127	113	94%	100%
`epiphany`	47	34	90%	100%
`evince`	470	37	62%	100%
`gedit`	57	57	82%	100%
`gnome-terminal`	83	58	86%	100%
`gnome-calculator`	32	32	100%	100%
`pidgin`	50	45	91%	100%

The remaining three applications: `evince`, `epiphany`, and `gnome-terminal` are good examples of applications which depend on multiple graphical libraries. While our GTK+ instrumentation worked as intended for these applications, hence some coverage of their interface, these applications rely on additional graphics libraries to handle portions of their functionality. For example, while browsing the web with `epiphany` our instrumentation was able to capture the window title, items on the menu bar, and the URL; however, it was unable to read any of the text from within the displayed web pages. This is because `epiphany` is built upon the WebKit rendering engine [44], a library which we have not instrumented. However, it is worth noting if we were to instrument WebKit, we would then be able to support all applications utilizing its rendering features as well as GTK+.

Similarly, both `evince` and `gnome-terminal` use additional libraries to support their functionality. The pdf viewer `evince` relies on the Poppler PDF rendering library [45] and `gnome-terminal` relies on the Gnome Virtual Terminal Emulator [46] library to power the terminal display. Just as with WebKit, if we were to instrument these libraries, we would have text extraction for all applications which used them. Interestingly, despite these other libraries, when text is highlighted and copied to the clipboard, we are able to extract the text through GTK+ as these applications utilize the clipboard API of GTK+ instead of the other libraries. To capture all text, from all applications, an organization deploying our approach would need to instrument all of the graphical libraries utilized by applications within the organization. While this may seem like a large and onerous task to undertake, there are significantly fewer graphical libraries than there are applications. As we have shown, just instrumenting a single graphical library leads to successful text extraction from many applications; by instrumenting just a few additional libraries, the number of supported applications would increase dramatically.

Diving a little deeper into the details, we review the effectiveness of our approach within the `pidgin` instant messaging client. Not only was our approach able to successfully extracted text such as "Options", "Join a Chat?," and "Mute Sounds" from the main window of the application, we also were able to extract text from the conversation windows. We further explored this by establishing an instant messaging conversation between two participant hosts (with one having our GTK+ modifications) and created a new chat. We sent a few messages back and forth and were able to observe a transcript of the chat in our extraction logs. We include an except from the log below:

```
338534.758328743 32221 gtk_text_buffer_insert 0xb962f6c0 "(12:59:07 PM) "
338534.759455910 32221 gtk_text_buffer_insert 0xb962f6c0 "d:"
338534.760105082 32221 gtk_text_buffer_insert 0xb962f6c0 " "
338534.760726432 32221 gtk_text_buffer_insert 0xb962f6c0 "hello"
[ unrelated text from other apps omitted ]
338553.860663724 32221 gtk_text_buffer_insert 0xb962f6c0 "(12:59:16 PM) "
338553.872373970 32221 gtk_text_buffer_insert 0xb962f6c0 "CS:"
338553.873169254 32221 gtk_text_buffer_insert 0xb962f6c0 " "
338553.875125457 32221 gtk_text_buffer_insert 0xb962f6c0 "howdy"
```

As our approach creates a new log entry for each invocation of the GTK+ public API functions, we are able to see the processing that `pidgin` is performing on incoming messages. First, we see the pidgin prepended timestamp, then the author of the message ("d" and "CS"), followed by the actual message. Additionally, we observed extracted text such as "CS has stopped typing" which indicates we are successfully detecting even the typing notifications.

As another example, we also tested our extraction capabilities within the `gedit` text editor. While using the application we observed the successful extraction of the text present on the main window, buttons, menu-items, and even the user-editable text area. In the excerpt from our logging file below, we can see gedit process the load of the file "file_test2." Quickly afterwards we extract the full path of the file as `gedit` updates a status blurb in its main window followed by the insertion of the file contents "Hello World!" into the editable text space. Additionally, we are able to observe the location of the text cursor through the extraction of the current location indicator present in the main `gedit` window.

```
340030.490457339 457 gtk_label_set_text 0x9f4d730 "file_test2"
340030.494264872 457 gtk_label_set_text 0x9f13b98 "Loading
    file '/home/cyber/file_test2'..."
340030.538422267 457 gtk_text_buffer_insert 0x9ef65c0 "Hello World\!"
340030.538854912 457 gtk_label_set_text 0x9f13d08 "  Ln 2, Col 1"
```

When considering the deployment of our approach, a potential concern by an organization may be the prospect of repeatedly instrumenting the graphical libraries as updates occur. Intuitively, we would expect most if not all of the original instrumentation to be directly portable to updated code as with large libraries code does not change drastically very frequently. To confirm this intuition we explored the viability of updating our GTK+ code to run on the latest release of Ubuntu: version 14.04. In a similar effort as our original approach on 12.10, we aimed to use the exact version of GTK+ packaged with the distribution; in the case of Ubuntu 14.04, the GTK+ versions are 2.24.23 and 3.10.8. Since our original approach favored instrumentation in the GTK+3 version of the library, we again focus on the newer library. We generated a list of all the locations where we instrumented the original GTK+3 library, and then manually copied over the instrumentation code from the 3.6.0 version (Ubuntu 12.10) to the 3.10.8 version (Ubuntu 14.04). After only a few hours of work we had a fully operational and instrumented copy of GTK+3 working and extracting text.

While two of our original instrumentation hooks were no longer necessary due to code refactoring, the remaining code was equivalent. Because we simply migrated our existing instrumentation over to the new version, it is possible that we miss text handled in newly introduced API functions. To compensate for this, we could re-perform our systematic analysis of the code base to note any differences between the analysis of the original 3.6.0 code base and our updated 3.10.8 version. Alternatively, we could simply review the GTK+3 release notes for any newly introduced functions. However, for a minimal amount of effort we already achieved extraction of a vast majority of the text passing through the library. To make the process even quicker, organizations could simply create a program which is able to automatically insert the instrumentation code in the correct places and only consult a developer if a refactor in the codebase is detected.

6.3.2 Performance of Data Extraction

We assess the performance overhead of our approach by determining the amount of time our code requires to extract the data required and save it to a file. We focus on this measurement as any additional time taken to update the graphics on the screen can cause update lag which would be noticable by a user. For our approach to be viable, we must have low enough overhead to remain unnoticed by the user. To quantitatively measure the time we spend processing within GTK+ we employ the `clock_gettime` function to record nanosecond resolution at the beginning our processing, and again after the log file is closed. We save the difference between the two recorded times to a separate performance log file.

With the code for performance measurements in place, we exercised our GTK+ 3.10.8 implementation on a Ubuntu 14.04 virtual machine allocated with 2GB of RAM and four 2.6GHz cores. To generate data, we ran a number of applications, expanded their menus, opened up dialog boxes, and then quit the application. In all, we logged a total of 36,409 performance results. After some analysis we removed 215 outliers from our statistical analysis as they were more than two standard deviations from the mean. We attribute the existence of these outliers to operating system scheduling delays such as queued context switches. The statistics from our results can be seen in Table 7.

Table 7: Overhead time for our instrumentation to log a GTK+ API call in μs as seen through 36,409 data points. 215 points excluded from the statistics for being two times the standard deviation greater than the mean.

Minimum	Median	Average	95th Percentile	Maximum	Standard Deviation
44.24	71.79	100.8	272.16	788.4	85.41

6.3.3 Using Data to Recreate the Operational Context

By combining our extracted text with our recorded hierachical information, or contexual awareness becomes more powerful than the sum of the two pieces. For example, by leveraging the recorded parent-child relationships we can construct a graph representative of the widgets created by an application. By traversing this graph, we can learn more about the location of text than otherwise would be possible. For example, in the excerpt of our log below we are able to trace the assignment of text all the way up to the enclosing window by connecting objects keyed by their address in memory. We represent the log entries below in logical order, from child to parent, instead of temporary.

```
338524.036541612 32249 gtk_label_set_text 0x84b34d0 "Print"
338524.036960916 32249 0x84bacb8 0x84b34d0 gtk_widget_set_parent
338524.037202966 32249 0x84bb168 0x84bacb8 gtk_container_add
338524.033204254 32249 0x84625a0 0x84bb168 gtk_container_add
338563.405579625 32249 (nil)     0x84625a0 gtk_widget_set_parent_window
```

When our instrumentation records information relating to the parent-child relationships, we also make note of the function we are receiving data from. In doing so we gain even more contextual information about the objects being handled. For example, when an application requests that GTK+ set the text of a drop-down menu it calls the `gtk_menu_item_set_text` API function. By simply looking at the name of the function, we can reason that the object associated with that API invocation is a GTK+ menu widget.

In Figure 3 we give an example of how our contextual information can be used to construct a graph representing the user interface. Labeled nodes represent an object with the assigned text show as the node label. Black nodes are one or more connected widgets without text assigned to them. By inspecting the contents of the graph and the relative distance between the objects with text, we are able to reason about the proximity of an object to another. This could be be used as a quantitative metric of the relationship between textual objects on the screen and could lead towards the development of security policies or other studies on the layout of applications. For example, extracting the word “print” from the on-screen interface may in of itself be innocuous; however, we could alert security systems if we see the word “print” associated with a button, and detect known business-confidential information on the screen. A security system may then use our information to decide any subsequent print attempt from that host should be denied.

Using the extracted data, we can also begin to make statements about the behavior of applications as recorded by the Linux Auditing Subsystem. For example, as shown in the abbreviated log

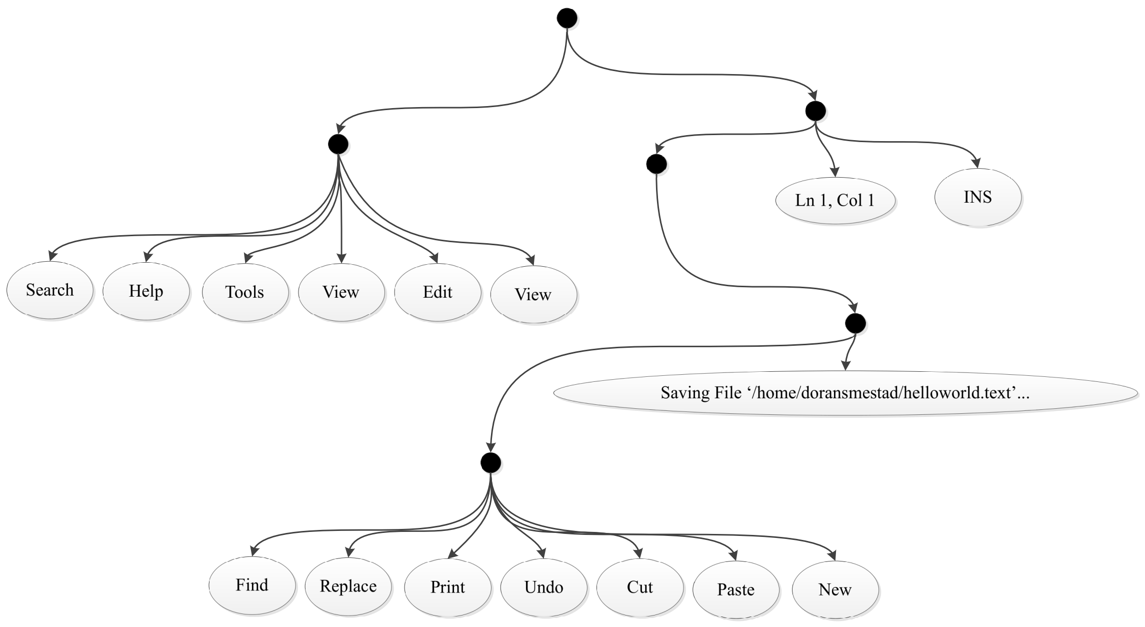

Figure 3: Graph depicting a subset of the relationship of text elements in the GTK+ container hierarchy for the `gedit` application.

excerpt below, we observe the assignment of the text "Save" to a label which is then assigned to a button.

```
1423590487.994 gtk_label_set_text_with_mnemonic 0x9cfc170 19244
    "/usr/bin/gedit" "_Save"
1423590487.994 19244 0x9cfc0b8 0x9cfc170 gtk_widget_add_mnemonic_label
1423590492.490 19244 0x9cfc0b8 EVENT-ButtonPress
1423590492.742:2458703 type=PATH : item=0 inode=590808 mode=file,664
    name=/home/test_user/test_document
1423590492.742:2458703 : cwd=/home/scapegoat
1423590492.742:2458703 type=SYSCALL : arch=i386 syscall=open success=yes
    exit=11 pid=19244
```

At some point in the future, this button is activated by the user which causes a chain of system events to occur. According to the Linux Auditing System, `gedit` opens, writes to, and then closes the file "home/test_user/test_document." Using our extracted text and generated context graph we can provide background information indicating what the user observed on the screen, and then most importantly, identify the cause of the system actions: a button press by the user.

7 Fusion of the Sensors

To evaluate the effectiveness of our sensors in describing user interaction on a system, we interacted with our system to perform a series of tasks which an end user would normally do. In this section we will describe our testing methodology and the actions we performed while our sensors extracted information from the interface. We then evaluate how well our approach performed by reviewing the performance data of each sensor and how well our sensors are able to describe the user activity which occurred on the system.

7.1 Testing Methodology

When creating our methodology, we aim to fully exercise all of our sensors and also record the actions all processes running under our testing user. We prepared our system by first creating a new user, "demo," under which the test would be run. To provide us with some files to interact with, we placed a two minute long video clip of an open-source movie [47], and a text file named "helloworld.txt" containing simply "HelloworldFile" as its contents. In order to monitor the activity of processes running under the demo user, we use the Linux Auditing Subsystem to monitor system call invocations. To configure the auditing subsystem towards our needs, we edited the configuration file (`/etc/audit/auditd.conf`) to instruct the auditing system to perform high-performance buffered logging. We then edited the auditing rules file (`/etc/audit/audit.rules`) and inserted the following commands (indented lines are a continuation of the previous line).

```
-a exit,never -F arch=b32 -F euid=1001 -F dir=/home/doransmestad/ -S open
     -S close -S fork -S execve -S clone -S socketcall
-a exit,never -F arch=b64 -F euid=1001 -F dir=/home/doransmestad/ -S open
     -S close -S fork -S execve -S clone
-a exit,always -F arch=b32 -F euid=1001 -S open -S close -S fork -S execve
     -S clone -S socketcall
-a exit,always -F arch=b64 -F euid=1001 -S open -S close -S fork -S execve
     -S clone
```

At a high level, the commands indicate to the Linux Auditing Subsystem that we wish to log the following system calls: open, close, fork, execve, clone, and socketcall. We selected this subset of the system calls to keep the volume of the log low while giving us insight into file and network operations processes are performing. Given the power of the auditing subsystem, we need to be precise when indicating what we wish to log. As such, whenever auditing system calls, the audit subsystem needs to know if we wish to instrument the 32 bit or 64 bit version of the calls. To be thorough, we indicate the to system that we wish to log both. This requires two very similar commands which we will refer to as a 'pair'. The first pair of commands tells the auditing subsystem to exclude the home directory where our sensors save their log data, no need to audit our own logs. The second pair of commands indicate that we wish to record invocations of the aforementioned system calls only when the user owning the process is our demo user (whose uid is 1001). We note there is slight difference between these pairs as the 64 bit system calls have no socketcall listing and therefore is not monitored for 64 bit processes.

With the foundation for our test laid, we test our implementation by performing the following tasks on a Ubuntu Desktop 14.04 virtual machine with 2GB of RAM and four 2.6GHz CPU cores:

1. Clear out our sensor's logs
2. Reboot the virtual machine
3. Start wall-clock stopwatch
4. Using the graphical interface, authenticate as our demo user
5. Move the mouse cursor around the screen, dragging and clicking objects on the desktop
6. Launch `gnome-terminal`, list a few directories, then launch applications
7. Launch the default Ubuntu video player, `totem`, and watch our two minute video
8. Launch the Chromium Browser and read two web-based news articles
9. Launch the text-editor `gedit`
10. Using `gedit`, open our "Helloworld.txt" file and type a few paragraphs of text
11. Collect the log files

7.2 Evaluation

We evaluate the results of our test by reviewing the events recorded in our sensor's logs then by analyzing the performance overhead of our approach. We perform simple aggregation and analysis with a few scripts we wrote to provide insight into the recorded data, and by extension, the interactions occurring with the user interface.

7.2.1 Analysis of Sensor Data

We begin by studying the log data from our Kernel sensor to obtain the reported consumers of input from the input devices. Our kernel sensor recorded 7,960 events related to the capture of the user input devices. Interestingly the X Windowing System was not the only process which attempted an open to of the devices. We saw evidence indicating that "systemd-logind", "acpid", "plymouthd", and two shell scripts also attempted to open the keyboard, but never subsequently read any input. We discount them from further consideration. As expected, and can be viewed in the log except below, our kernel sensor indicates that our X Windowing System, noted as "X," opened the input devices.

```
Open pid:1136 fd:7 exe:X file:/dev/tty7 perf:387
[ unrelated log entries hidden ]
Open pid:1136 fd:10 exe:X file:/dev/input/mice perf:1052
Open pid:1136 fd:10 exe:X file:/dev/input/mice perf:655
Open pid:1136 fd:10 exe:X file:/dev/input/mice perf:997
```

Upon further inspection of the logs in an aggregate form, we are able to note that the X Windowing System invoked the `read` system call 7427 times to grab input from the keyboard and mouse. To further separate the out the details it can also be observed from the aggregate data that X consumed 3,574 bytes from file descriptor 7 (keyboard input) and 20,347 bytes from the file descriptor 10 (mice input). Based upon our measurements when evaluating the kernel sensor (Section 4.3.2), we estimate there were approximately 1,800 keyboard presses and at least a few minutes of mouse input.

To verify this claim, we move on to inspecting the results of the X11 sensor. During our tests the X11 sensor logged 70,618 events. We aggregated these events and distilled them down to 622 entries describing the processes which connects with the X Windowing System interface. Surprisingly, there were a number of applications which established the connection to the X Windowing System, but then never used the connection to display data. To further filter the aggregate data we remove from consideration processes which we have: no more than twenty graphical outputs, zero mouse inputs, and zero keyboard inputs. This simple filter removed 614 processes which only connect to the XServer without actually causing user interaction beyond that.

With the non-interacting programs removed from consideration, we are left with the eight processes below. Four of these processes are the applications we intended to launch, and the other four are system processes responsible for powering the user interface. For brevity, we collapsed applications with multiple process identifiers and instead represent the sum of their events.

Client	PID	Output	Keyboard	Mouse	Output/Input Ratio
"nautilus"	3023	1054	10	412	40%
"bamfdaemon"	2101	2	57	616	336%
"compiz"	2902	47448	0	0	0%
"gnome-terminal"	3659	485	354	289	173%
"gedit"	9395	1502	2985	0	199%
"chromium-browse"	8891	4930	0	3743	76%
"totem"	3777	3351	16	400	12%
"unity-greeter"	1345	61	22	11	54%

We first notice that the summation of our X11's captured keyboard events compared to the kernel sensor's is nearly correct: 1722 keystrokes (3444 X11 events, one for KeyPress, one for KeyRelease) to the Kernel's approximately 1800 keystrokes. The difference we attribute to a combination of approximation error and the possibility of certain infrequent actions within X11 not being fully logged. Despite the difference, our ability to verify that our X11 sensor is correctly capturing nearly all, if not all, keystrokes reinforces the validity of the sensor in the X11 sensor.

Through the use of the X11 sensor data we are able to perform some reasoning concerning processes and their user-interaction. To aid our understanding, we categorize the processes into groupings based upon the way each process interacts with the user. Applications with high user input, or "input-heavy," are: `bamfdaemon`, `gnome-terminal`, `gedit`, `unity-greeter`, and `chromium-browser`. The only application with moderate user-input, or "input-balanced," is `nautilus`. Finally, the applications which mainly send information to the user, or "output-heavy," are: `compiz` and `totem`.

According to this categorization of processes, we are able to separate the applications which simply appear on the screen from applications which may be actively engaged in completing user-driven tasks. In doing so we are able to say with certainty that the user acknowledges the application and drives actions the processes are performing. Unfortunately, our X11 sensor has no way of knowing what the user is actually observing within each window of the applications they use. While we are able to connect the user to the process, more information is needed in order to comprehend the user's intentions when interacting with a process' on-screen representation.

To gain further insight into what may be occurring within these processes, we employ our GTK+ sensor. With our instrumentation, we extract the information upon which the interface the user observes is constructed, and reconstruct it for automated reasoning. To best understand why a user initiated a certain action, we study actions with a catalyst. While our GTK+ sensor logs every bit of the user interface it processes, one way of reasoning about the data is to reconstruct the contextual information of an interface immediately after the user presses a button. Our instrumentation received 12 button presses from 2 processes. We discuss just one recorded button press: when a button was clicked within the `gedit` application leading to reading a file from the disk.

When running through the steps in our evaluation, we proceeded to open `gedit`, use `gedit`'s open-file window to browse to the text file we wished to edit, then open the file. By constructing a graph with the connected components of the graphical user interface at the time immediately surrounding the button press, we are able to inspect the state of the interface when the user decided to press the button. We present a representation of the reconstructed relationships between displayed objects in Table 8. The relationships indicates the button pressed, labeled "gtk-open", was placed on a "GtkFileChooserDialog." Nearby elements were set with the labels "Open" and "_Open" leading us to believe that the dialog the button press occurred was the save dialog.

Table 8: A flat depiction of the reconstructed GUI object graph as seen by the GTK+ sensor.

Distance from Button	Function Name	Text	Parenting Function
1	gtk_buildable_set_name	”dialog-action_area1”	gtk_widget_set_parent
2	gtk_buildable_set_name	”dialog-vbox1”	gtk_dialog_add_button
1	gtk_window_set_title	”Open”	gtk_dialog_add_button
	gtk_buildable_set_name	”GtkFileChooserDialog”	gtk_dialog_add_button
2	gtk_button_new_with_label	”gtk-cancel”	gtk_widget_add_mnemonic_label
0	gtk_dialog_add_button	”gtk-open”	
	gtk_button_new_with_label	”gtk-open”	
	gtk_button_set_label	”gtk-open”	
1	gtk_widget_create_pango_layout	”Open”	gtk_widget_add_mnemonic_label
	gtk_label_new_with_mnemonic	”_Open”	gtk_widget_add_mnemonic_label
	gtk_label_set_text_with_mnemonic	”_Open”	gtk_widget_add_mnemonic_label
1	gtk_widget_create_pango_layout	”home”	gtk_widget_set_parent
	gtk_label_set_text	”home”	gtk_widget_set_parent
	gtk_label_new_with_mnemonic	”gtk-open”	gtk_widget_set_parent
	gtk_label_set_text_with_mnemonic	”gtk-open”	gtk_widget_set_parent
	gtk_label_new	”gtk-open”	gtk_widget_set_parent

With this information in hand, we can then temporally match the detection of a file-open dialog to system audit logs handling the opening and reading from a particular file. In doing so, we note the presence of the following log entry:

```
type=SYSCALL msg=audit(1429852855.731:379764): arch=c000003e syscall=2
    success=yes exit=21 a0=1d6c7a0 a1=0 a2=0 a3=185aa80 items=1 ppid=3680
    pid=9422 auid=4294967295 uid=1001 gid=1001 euid=1001 suid=1001
    fsuid=1001 egid=1001 sgid=1001 fsgid=1001 tty=pts1 ses=4294967295
    comm="pool" exe="/usr/bin/gedit" key=(null)
type=CWD msg=audit(1429852855.731:379764):  cwd="/home/demo"
type=PATH msg=audit(1429852855.731:379764): item=0
    name="/home/demo/Desktop/helloworld.txt" inode=2327447
    dev=fc:00 mode=0100644 ouid=1001 ogid=1000 rdev=00:00 nametype=NORMAL
```

This audit log, though verbose, shows that the process with pid `9422` (gedit), opened the file `/home/demo/Desktop/helloworld.txt` approximately a tenth of a second after the button click was registered in `gedit`.

As our GTK+ sensor is run completely under the user's privileges (outside of the rusted computing base), we wish to verify that the actions GTK+ is report are actually occurring. A few possible approaches in order to perform this correlation rely on determining the overlap of information, or obvious contradictions in such data, in which we are able to ensure we receive matching information. We suggest that a temporal relation between a GTK+ button press and the kernel and X11 sensors indicating a keyboard or mouse input are present. Additionally, we suggest a comparison between the quantitative amount information held by GTK+ be reconciled with X11's categorization of an applications input-type. That is, if GTK+ is reporting vast amount of context, but X11 never handled a screen presence, we would believe GTK+ did indeed render an interface but it never showed up on the screen.

Our system, by being able to provide the data able to connect the user all the way down to the system actions which result, can aid in our understanding of actions on the system. Our experimental validation of our system in this section shows that we are able to successfully identify the XServer as the only process reading raw input device information. We then show which processes have communicated with the XServer and have received copies of user-input as delivered from the XServer. With the list of applications in hand, we are then able to study the context of user actions as viewed by the user themselves through the automated extraction of text processed by the GTK+ graphics library. Utilizing this data to identify and even attribute actions on the system caused by the user become simpler. Instead of attempting to guess if a user caused an action on the system, any tool can leverage our approach to track which processes have accepted user input, and greatly reduce the pool of suspect processes during a security audit.

8 Discussion

Below we briefly discuss relevant aspects to the overall functionality of our approach, and some potential applications for our instrumentation of the GUI.

8.1 Disk Space

Running our three sensors at all times quickly requires a significant amount of disk space. For example, over the 17 minute capturing of data on our machines resulted in nearly 4MB of GTK+ logs, 12MB of X11 logs, and 1MB of kernel logs. This will lead to an aggregate log size of over 1.5GBs in just a 24-hour period. While our performance overheads are minimal, our disk presence is quite significant. We propose that any potential deployers make efforts to keep their disks from completely filling up. Alternative methods such as storing all log information on a network server, database server, or automatically compressed partition would also be able to provide benefits.

8.2 Application Domains

As our approach is a generic way to instrument the graphical user interface, our data can be used to aid in a number of areas. We explore a few possibilities below.

8.2.1 Usability Studies

To test the effectiveness of graphical interface layouts to communicate their meaning to an end-user, GUI designers usually run usability studies. There are three main types of usability studies: empirical, formal, and informally [48]. The empirical studies bring in human subjects and ask that they navigate through some designed interface while monitored. Formal and informal studies are cheaper solutions needing only a single expert or automated system to judge the viability of specific designs based upon prior experience [48]. While our approach could be leveraged in automated studies to determine the link-distance of various GUI objects, we focus our discussion on our instrumentation's ability to record user actions.

One of the main reasons that empirical usability studies are avoided is because of their high cost and challenges in recruiting a sufficiently diverse set of users to study. A few products, such as Google Chrome and Firefox attempt to resolve this by packaging their own sets of instrumentation and simply allowing the user to choose to self-report data. These forms of usability studies are not as formally defined and users may take whichever path through the program they wish, at any time. Nonetheless, they provide valuable feedback for the developers of web-browsers so that they can continue improving their product. Using our already existing sensors, we can provide this insight for the entire operating system, including individual programs which would otherwise not have such instrumentation.

Furthermore, we can extend the idea of these usability studies to include organizational wide metrics for application usage. For example, a business manager may wish to know if the costly licensed software is actively being used by the organization's employees or not. It may very well be that employees are still using Microsoft PowerPoint to create diagrams instead of using Microsoft Visio. Or, employees do use Visio, but only for simplistic drawings which could easily be powered by an open-source alternative. As another example, an organization would like to know how much time their employees spend in their web browsers instead of accomplishing business tasks: through our existing sensors we can quantitatively state the amount of user input to web browsers which correlates to the time spent in the application.

8.2.2 Security

The identification of protection against malware and other system security threats depends upon having actionable intelligence. Whether that is an anti-virus signature or an anomaly-based intrusion detection system, each requires data sets to process. Our sensors can be used directly for these purposes. By tracking which processes are interacted with by the user, we can inform security systems which applications have earned some level of 'endorsement' from the user. That is, an application which goes through the work of creating a user-interface and interacting with the user can be differentiated from a hidden process which performs actions without the user knowing. While there are categorizes of malware, just as a trojan-horse, which would be passed over by this naive approach, being able to provide the additional insight to a process' actions can aid in the detection of abnormal behaviors.

Our sensors could further be used within a network control paradigm. For example, if the host is operating within a software defined network (SDN) which deploys access control nodes on each end-host our data could become a valuable resource when writing policies. With our data a network administrator could require that a process interact with the user prior to allowing the outbound network traffic. While such a policy does require an end-host client for SDNs to communicate the data up to the network controller, contemporary work by Douglas MacFarland is already implementing such an approach [49]. A number of other security-focused applications are possible, but our main contribution to the security space is our data available for analytic by existing security solutions. More data allows for these system to produce better decisions, leading to an increase in security.

8.3 Ethical and Legal Considerations

As is true with any monitoring and data-collection system, ethical and legal considerations must be taken into account. Our sensors examine the information flow between various points in the graphical user interface and in the case of GTK+ actively extract information from the screen. Due to our presence in privileged areas, our sensors are able to intercept all user keystrokes and screen content. As such, we are able to log any password the user types and also extract possibly sensitive material from the screen such as a social security number, bank account information, or private communications. However, any other security solution installed will also be able to retrieve this sensitive information. In fact, commonly installed anti-virus solutions must open, read, process, and potentially report upon all files residing upon the host system.

Existing organizations and enterprises already understand these potentially problematic privacy issues and respond by requiring all of their employees to explicitly acknowledge they may be monitored while using organizational equipment. These warnings are usually presented upon login through user agreements or through dissemination of agreements in which employees are required to waive their privacy rights. Under these conditions, we believe that the use of our sensors is both legal and ethical. However, as with any tool potentially able to breach privacy boundaries, it is up to the deployer of our sensors to ensure their ethical use.

9 Future Work

Through the implementation and evaluation of our sensors we identified areas where our approach can be improved and expanded upon in the future. We discuss these areas below.

9.1 Instrument Additional Graphics Libraries

To support the extraction of context from additionally applications instrumentation of more graphics libraries is required. While our current GTK+ sensor works well in extracting context from the processes which employ its services, additional applications such as the PDF viewer `evince` use alternative libraries for the rending of the PDF contents. We specifically focus on the web browsing libraries such as `WebKit` and `Gecko` as the best next step. As web browsing has become ubiquitous over the past years, being able to see into the user's browser will greatly increase our coverage of common applications used by the end-user.

9.2 Refine the X11 Sensor

Below we discuss a few aspects in which improvements can be made to our X11 sensor.

9.2.1 Tracing the Event Data Flow

In creating the X11 Sensor, we place instrumentation the best place possible to capture both input events and output requests as they passed through the system. In doing so, we were able to capture nearly all of the events with perfect accuracy. However, there are occasional cases where our instrumentation is not fully complete. For example, when a process requests a 'grab,' to claim exclusive access to the keyboard and/or mouse our instrumentation is not fully able to log the flow of events. In future work, we aim to fully trace the flow of events and locate the infrequent flows which circumvent our instrumentation. Once identified, we aim to refactor our code to remain efficient and expedient while ensuring all events are fully captured.

9.2.2 Calculating Window Visibility

When determining whether or not a window is visible to the user we simply perform a few comparisons against the internal XServer data structures. If we are returned a value indicating that the window is only partially visible, our current code treats the window as if it were fully exposed. In the future, we want to handle this more carefully and calculate the actual window area exposed based upon the XServer's indication of the exposed portions. By doing so we will be able to further refine our quantification of a process' output, which will lead to a better and more accurate characterization of the user-process interactions on a machine.

9.3 Determining the User's Intention

With our three sensor system we are able to following user data from the kernel, to X11, and up to the GTK+ library. Our future work will look into extending our understanding to the users themselves. In particular, we want to study how users perceive an application on the screen and indicate an intention. This could be intending to the close the application by clicking an exit button, or indicating the desire to open a file by interacting with a file-open dialog. By establishing the intention held by the user we can begin to reason about process behavior which deviates from the expected actions to fulfill the intention. We believe such data would be highly useful towards the security field.

10 Conclusion

Modern computer systems are complex. As such, it can be difficult to understand the behavior of the system and why certain actions are occurring. In our approach, we proposed and implemented sensors capable of extracting information from the graphical user interface in order to identify the processes which interacted with the user. By knowing this information, we are able to enrich the situational awareness of the events occurring within a system and therefore provide insights towards the root cause of system events. We showed that our instrumentation was effective in characterizing different types of applications with an on-screen presence, and further we provided data towards the determination of user intentions. We also proved that obtaining the information from the user interface can be done in an efficient manner with minimal overhead allowing our approach to be deployed across an organization without causing any significant processing delays.

References

[1] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: using system-centric models for malware protection," in *ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 399–412.

[2] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[3] W. Cui, R. Katz, and W. Tan, “Design and implementation of an extrusion-based break-in detector for personal computers,” in *Computer Security Applications Conference, 21st Annual*, Dec 2005, pp. 10 pp.–370.

[4] J. Shirley and D. Evans, “The user is not the enemy: Fighting malware by tracking user intentions,” in *Proceedings of the 2008 Workshop on New Security Paradigms*, ser. NSPW ’08. New York, NY, USA: ACM, 2008, pp. 33–45. [Online]. Available: http://doi.acm.org/10.1145/1595676.1595683

[5] Mozilla Developers, “Telemetry faq,” https://wiki.mozilla.org/Telemetry/FAQ, Sept 2014.

[6] Canonical Ltd., “Ubuntu,” http://www.ubuntu.com, Apr 2015.

[7] X.Org Foundation, “X.org,” http://www.x.org/wiki/, Apr 2015.

[8] The GTK+ Team, “Gimp toolkit,” http://www.gtk.org, Apr 2015.

[9] RedHat Developers, “Linux auditing subsystem,” https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html, Apr 2015.

[10] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” Proceedings of the IEEE, 1975.

[11] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *ACM Conference on Computer and Communications Security*, 2007, pp. 116–127. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315261

[12] K. Borders, X. Zhao, and A. Prakash, “Siren: catching evasive malware,” in *2006 IEEE Symposium on Security and Privacy*, May 2006, pp. 6 pp.–85.

[13] M. Salem, S. Hershkop, and S. Stolfo, “A survey of insider attack detection research,” *Insider Attack and Cyber Security*, pp. 69–90, 2008.

[14] W. Ju and Y. Vardi, “A hybrid high-order markov chain model for computer intrusion detection,” *Journal of Computational and Graphical Statistics*, vol. 10, no. 2, pp. 277–295, 2001.

[15] R. Maxion and T. Townsend, “Masquerade detection using truncated command lines,” in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on Dependable Systems and Networks.* IEEE, 2002, pp. 219–228.

[16] K. Yung, “Using self-consistent naive-bayes to detect masquerades,” *Advances in Knowledge Discovery and Data Mining*, pp. 329–340, 2004.

[17] K. Wang and S. Stolfo, “One-class training for masquerade detection,” in *Workshop on Data Mining for Computer Security, Melbourne, Florida*, 2003, pp. 10–19.

[18] B. Szymanski and Y. Zhang, "Recursive data mining for masquerade detection and author identification," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*. IEEE, 2004, pp. 424–431.

[19] J. Seo and S. Cha, "Masquerade detection based on svm and sequence-based user commands profile," in *ACM Symposium on Information, Computer and Communications Security*. ACM, 2007, pp. 398–400.

[20] Apple Incorporated, "iOS security," https://www.apple.com/privacy/docs/iOS_Security_Guide_Oct_2014.pdf, October 2014.

[21] Android Developers, "Android security overview," https://source.android.com/devices/tech/security/.

[22] P. F. Wilbur and T. Deshane, "Johnny can drag and drop: Determining user intent through traditional interactions to improve desktop security," in *Symposium on Computer Human Interaction for the Management of Information Technology*, ser. CHiMiT '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/1873561.1873565

[23] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *2012 IEEE Symposium on Security and Privacy (SP)*, May 2012, pp. 224–238.

[24] Z. B. Li, P. R. Kane, and C. A. Shue, "Insider threat detection with text libraries," Worcester Polytechnic Institute, Tech. Rep. eProject 012414-115432, December 2013.

[25] D. N. Muchene, K. Luli, and C. A. Shue, "The big picture: Using desktop imagery for detection of insider threats," Worcester Polytechnic Institute, Tech. Rep. eProject 135836, December 2012.

[26] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-Bot: Improving service availability in the face of botnet attacks." in *NSDI*, vol. 9, 2009, pp. 307–320.

[27] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 440–450.

[28] H. Zhang, W. Banick, D. Yao, and N. Ramakrishnan, "User intention-based traffic dependence analysis for anomaly detection," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*. IEEE, 2012, pp. 104–112.

[29] K. Xu, D. Yao, Q. Ma, and A. Crowell, "Detecting infection onset with behavior-based policies," in *Network and System Security (NSS), 2011 5th International Conference on*, Sept 2011, pp. 57–64.

[30] H. Zhang, W. Banick, D. Yao, and N. Ramakrishnan, "User intention-based traffic dependence analysis for anomaly detection," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, May 2012, pp. 104–112.

[31] W. N. Bhukya, S. K. Kommuru, and A. Negi, "Masquerade detection based upon gui user profiling in linux systems," in *Advances in Computer Science–ASIAN 2007. Computer and Network Security*. Springer, 2007, pp. 228–239.

[32] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[33] QEMU Developers, "QEMU," http://wiki.qemu.org/Main_Page, Apr 2014.

[34] A. Cox, "Linux allocated devices," https://www.kernel.org/doc/Documentation/devices.txt, Apr 2009.

[35] Microsoft, "Snap - microsoft windows," http://windows.microsoft.com/en-us/windows7/products/features/snap, Apr 2015.

[36] Compiz Developers, "Compiz," http://www.compiz.org, 2015 Apr.

[37] K. Packard, "X display manager control protocol," http://www.x.org/releases/X11R7.6/doc/libXdmcp/xdmcp.html, Apr 2015.

[38] Ubuntu Developers, "Lightdm," https://wiki.ubuntu.com/LightDM, Apr 2015.

[39] Canonical Ltd., "Release end of life," http://www.ubuntu.com/info/release-end-of-life, Apr 2015.

[40] X.Org Developers, "Building the X window system," http://www.x.org/wiki/Building_the_X_Window_System/#index9h3, Apr 2015.

[41] P. Hutterer, D. Stone, and C. Douglas, "The X input extension 2.x," ftp://www.x.org/pub/xorg/current/doc/inputproto/XI2proto.txt, Mar 2012.

[42] M. Prokop, "ss: Another utility to investigate sockets," http://linux.die.net/man/8/ss, Apr 2015.

[43] GTK+ Developers, "Gtk+ 3," https://developer.gnome.org/gtk3/stable/GtkLabel.html#gtk-label-new-with-mnemonic, Apr 2015.

[44] WebKit Developers, "The webkit open source project," https://www.webkit.org, Apr 2015.

[45] Poppler Developers, "Poppler," http://poppler.freedesktop.org, Apr 2014.

[46] Gnome VTE Developers, "Gnome virtual terminal emulator," https://code.launchpad.net/vte, Apr 2015.

[47] Blender Foundation, "Big buck bunny," https://peach.blender.org/about/, Apr 2014.

[48] J. Nielsen, "Usability inspection methods," in *Conference Companion on Human Factors in Computing Systems*, ser. CHI '94. New York, NY, USA: ACM, 1994, pp. 413–414. [Online]. Available: http://doi.acm.org/10.1145/259963.260531

[49] D. C. MacFarland, "Exploring host-based software defined networking and its applications," Master's thesis, Worcester Polytechnic Institute, May 2015.