Can the User Help? Leveraging User Actions for Network Profiling

Zorigtbaatar Chuluundorj*, Curtis R. Taylor†, Robert J. Walls*, Craig A. Shue*
*Computer Science Department, Worcester Polytechnic Institute, Worcester, MA, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
{zchuluundorj, rjwalls, cshue}@wpi.edu, taylorcr@ornl.gov

***Abstract*—Enterprises have difficulty gaining insight into the steps preceding anomalous activity in end-user machines. Enterprises may log events to later reconstruct anomalies to gain insight and determine their causes. Unfortunately, most logs are low-level and lack contextual information, making manual inspection arduous. Accordingly, enterprises may fail to promptly respond to anomalies, leading to outages or security breaches.**

To help these enterprises, we monitor and log each user's interactions with the machine's user interface (UI) and link them to the resulting network flows. We design, implement, and evaluate an SDN system, called HARBINGER, for the Microsoft Windows OS that provides user activity context for flows.

Enterprises can use the context we gather to complement traditional analysis. We explore how HARBINGER can help differentiate normal and abnormal network traffic. While IP or DNS host name profiling can have error rates between 29%-38% for URL-based traffic, UI-aware sensors can reduce such errors to 0.2%. We further find that with the help of user action tracking, we can detect errant network traffic 99.1% of the time in our tests. HARBINGERhas good performance, introducing less than 6 milliseconds of delay in 95% of new network flows.

I. INTRODUCTION

When an end user's machine exhibits anomalous behavior, enterprises often use log files to understand the issue. They try to understand the causal chain of events that lead to anomalous behavior. Log files are essential for IT system management and monitoring and can help expedite problem diagnosis and reduce resource usage and infrastructure costs. They can also reveal the early stages of potential security breaches. Unfortunately, these log files tend to be low-level and can overwhelm individuals with unrelated data. As a result, enterprises may lack the clarity needed for effective response.

Given these challenges, we seek to improve log utility by contextualizing it and linking it to end-user behaviors. We ask the following research questions: *To what extent can UI data and user actions be fused with low-level resource requests? What impact can such context have in isolating anomalous network behavior? Can such data reveal malicious activity?*

To give enterprises more in-depth insight, we monitor the user interface (UI). We provide instrumentation that correlates fine-grained UI data with network traffic without requiring application-specific instrumentation. We design and build the HARBINGER software-defined network (SDN) system to acquire context that signals forthcoming network connections to detect abnormal behavior that could identify potential configuration errors, failing components, or malicious acts.

Our approach is to link the host operating context with the resulting network flows to enable dynamic network traffic profiling. HARBINGER fuses traditional network knowledge, such as a flow header, with information about the application, such as the account username running the program and its executable path. To understand human behavior, which is a significant source of non-determinism in a system, HARBINGER further collects and links end-user interactions via keystrokes and mouse clicks with graphical user interface (UI) context, such as buttons, windows, and other UI widgets. By doing so, HARBINGER allows organizations to distinguish what events are user-driven from those that are automated.

With HARBINGER, we explore the feasibility of designing and building a universal fine-grained UI monitor with limited application-specific instrumentation. We then explore the extent to which the acquired UI information can improve profiling network traffic and the system's utility in an enterprise. Our novelty lies in the use of fine-grained instrumentation that is tightly linked to low-level resource requests in an application-agnostic manner. In doing so, we contribute the following:

- **UI Data Acquisition Design Challenges:** We describe our design goals and challenges in implementing such a design (Section III). We create a trace formalism to associate UI precursors with network actions.
- **System Design and Implementation:** We design and build HARBINGER, for the Microsoft Windows operating system, which fuses UI data with network traffic. We describe and implement HARBINGER's UI data acquisition and use software-defined networking (SDN) to share data with a centralized SDN controller (Section IV).
- **Evaluation of UI Data Application in Profiling Network Traffic:** We compare traditional profiling with that enabled by UI sensors (Section V). We find that UI sensors can reduce misclassification in URL-based workflows while also identifying errant traffic that went undetected by the IP and DNS sensors.
- **Evaluation of Performance:** We evaluate the performance of HARBINGER (Section VI) and find it affects only the first exchange in each flow and adds an end-to-end delay of less than 6ms.

II. Background and Related Work

Suneetha *et al.* [1] used web log filing of user behavior to improve the experience. UI logging may offer similar value.

System logs and system call analysis have been used to reconstruct a user's actions based on low-level events that result from that action [2], [3]. A higher-level event, such as opening a word processing application, may produce a set of low-level features (log or system call entries). Such techniques may incur high overhead and have limitations with interleaved events [4]. Csight [5] focused on mining existing concurrent systems execution logs to infer a concise and accurate model of a system's behavior for debugging purposes. Yu *et al.* [6] developed CloudSeer for workflow monitoring to detect deviations from normal behavior in cloud services. CloudSeer runs in cloud infrastructure to build automaton based on normal executions logs. Du *et al.* [7] developed DeepLog, which models system logs as a natural language sequence with deep neural modeling to detect anomalies. We can aid these techniques by capturing user actions, which constitute a significant source of non-determinism in a system, relating them to the low-level activities these systems use.

Log files can be used for security purposes, such as intrusion detection. Handigol *et al.* [8] developed NetSight, a packet capture utility, to buid a network debugger and profiler. They argue that packet history can reveal the root cause of network failures. Our approach provides the historical activity of UI workflows that may provide insight into an application's abnormal behavior from a higher-level historical perspective.

BINDER [9] observes that malware often avoids user interaction, so BINDER allows all of an application's network traffic if it immediately follows a user input. Since BINDER does not link traffic destinations to specific user interactions, its broad authorization enables mimicry attacks [10] in which malware times its traffic to evade detection. Kwon *et al.* [11] used more detailed user activity instrumentation to detect botnet applications, but the approach is still vulnerable to mimicry attacks due to limited understanding of an application's interface. HoNe [12] correlates network traffic with metadata processes but lacks visibility into user actions. In Gyrus [13], a VM hypervisor inserts a secure text overlay over untrusted applications and ensures entered data matches what is subsequently transmitted on the network. In contrast to these approaches, our tool can link user actions to specific network destinations, limiting adversaries' opportunities to blend in.

Bhukya *et al.* [14] profiles user typing and clicking behavior to detect when one end-user attempts to impersonate another. Others have used custom user interfaces to tighten file access control [15], [16]. Shirley *et al.* [17] track file creation provenance and user behavior to determine when applications interact with files from other applications. However, they are application-specific and do not examine network traffic.

Our work extends the literature by 1) constructing a system that fuses detailed UI events with network flows and 2) incorporating this data in profiling network traffic.

III. UI Data Acquisition and Formalism

A *UI-aware system* collects the user's interaction with each application's graphical UI on the monitored host. The collected information must be precise (i.e., fine-grained and specific to the user's goals) and accurate. It can be challenging to link a user's UI actions to lower-level application behavior, such as a particular network request. While application-specific instrumentation may reveal causal links, such efforts are time-consuming and arduous. Instead, we explore the use of temporal correlations to infer causal links.

A. UI and Network Connection Formalism

To represent the union of UI workflow events and a set of resulting network connections, we define a *trace* formalism. We denote a trace based on the application, UI control flow elements, a set of inputs, and corresponding connections. A trace can be denoted as: $\{\texttt{Application}_1\texttt{, (a, b, c, d), inputs}\} \rightarrow \{\ \texttt{connection}_1\texttt{:time}_1\texttt{, connection}_2\texttt{:time}_2\ \}$. In this example, a given application has a sequence of connected UI widgets (a, b, c, d) that are predictive of a set of network connections that occur within a specified amount of time. This trace represents a UI workflow and its ensuing network connection.

Figure 1 shows a HARBINGER trace where an end-user types and clicks a hyperlink in Microsoft Word. The user types a 34 character string, which included the text "https://example.com," which Microsoft Word automatically converts into a hyperlink. The trace includes user inputs in terms of keystrokes and mouse clicks, the structure of the GUI hierarchy (including the Word title bar), information about the application path to the process executable, the user running the application, and details about the network connection, including addresses, ports, and transport protocol fields. The record shows the destination host name based on a prior DNS response that had an `A` record matching the destination IP address. This trace shows the origin of the network flow.

Traces may be self-referential, which is key to network profiling. When a user navigates via a URL, the URL itself will appear in the trace. The host name or IP address in the URL can thus be associated with the resulting network connection's destination. The policy rule in Figure 2 matches the request in Figure 1. This rule's trace uses regular expression and memory parenthesis notation to indicate that it matches a destination host name that appears as a hyperlink in the UI. The entry matches users clicking a hyperlink in Microsoft Word.

The trace formalism can support known automated behavior, such as activity based on timers, made to pre-programmed destinations (e.g., application update servers) or organization-configured destinations (e.g., email servers). In these cases, the set of UI events or user inputs may be omitted.

The trace formalism also supports variances in the resulting network behavior. For example, application layer caching may result in fewer network connections in some instances than others. Further, computational load or process activity may result in actions being delayed by hundreds of milliseconds. The time ranges can accommodate fluctuations when profiling.

```
Date: Sept. 10, 2018 at 9:54:17am
Source: [ANONYMIZED IP], port 49795
     Host Name: [ANONYMIZED]
Destination: 93.184.216.34, port 443
     Host Name: example.com
Protocol: TCP [SYN flag set]
Application: C:\Program Files (x86)\Microsoft
     Office\root\Office16\WINWORD.EXE
User: [ANONYMIZED_DOMAIN]\[ANONYMIZED USER]
Keystrokes (5 s, 15 s, 60 s, 3 m, 5 m):
     0, 34, 34, 34, 34
Mouse Click (5 s, 15 s, 60 s, 3 m, 5 m):
     5, 5, 6, 8, 8

User Interface (last event 231ms ago):
Name: Document1 - Word, Class: OpusApp,
|     Control: window
| Name: [none], Class: _WwF, Control: pane
| | Name: Document1, Class: _WwB,
| | |     Control: pane
| | | Name: Document1, Class: _WwG,
| | | |     Control: document
| | | | Name: Page 1, Control: page
| | | | | Name: Page 1 content, Control: edit
| | | | | | Name: https://example.com,
| | | | | |     Control: hyperlink
```

Fig. 1. Example trace from the HARBINGER system while using the Microsoft Word application. Keyboard and mouse inputs are binned by cumulative time intervals for the last 5 seconds, 15 seconds, 60 seconds, 3 minutes, and 5 minutes. The user interface entries begin with the relative number of milliseconds ago the element was created or refreshed in the GUI, followed by any label associated with the GUI element, an application-generated class for the GUI element, and a control type that describes the type of widget. The trace was formatted and annotated for readability.

```
Destination IP: *,
Destination Host: $1,
Destination Port: 443,
Protocol: TCP [SYN flag],
Application Path: C:\Program Files (x86)\
Microsoft Office\root\Office16\WINWORD.EXE,
Mouse Clicks/Keystrokes: 1+ in last 5 seconds,
GUI: .*https://([a-zA-Z0-9\-]+)/\s*, Control:
     hyperlink.*,
Action: Allow, label as 'normal'
```

Fig. 2. A profiling rule that matches the traffic from the trace in Figure 1. The GUI string is a regular expression and the destination host is the string extracted from the expression's memory parentheses in the GUI text.

IV. LINKING THE UI TO NETWORK FLOWS

Our goal with HARBINGER is to differentiate between normal and abnormal network traffic. This can help analysts identify the cause of anomalies in an enterprise environment, which may be associated with failures, misconfigurations, and attacks. We assume all traffic caused by known historic processes (e.g., software updates) and end-user actions to be normal. Analysts may still use other tools such as anomaly detection or intrusion detection systems, in addition to HARBINGER, to detect and analyze abnormal traffic.

To effectively unite network activity with end-user actions, we need a mechanism that monitors the application itself and a mechanism that monitors the network. In Figure 3, we depict these application and network monitors and how

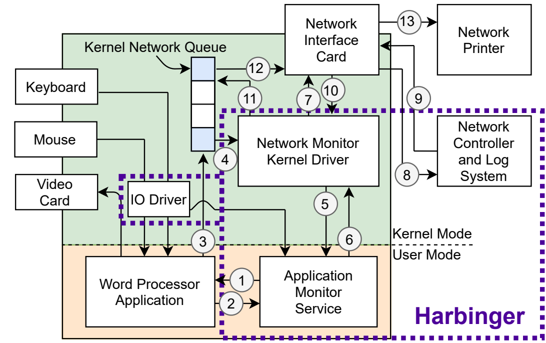

Fig. 3. System Overview. The dashed regions indicate the components introduced by HARBINGER. The numbered actions are described in Sections IV-A and IV-B.

they record user behavior and control network activity. We fuse this information and then log it off-system to support other profiling tools and incident response. The application and network monitors reside on the client machine and must be tailored specifically to the client OS. The policy engine can be run remotely on a centralized network controller to service multiple clients and operating systems.

The HARBINGER system is designed to enhance administrator understanding of a system and network. This may be used for security purposes (e.g., malware detection). To scope this work, we focus on the system's utility and defer formal security analyses to future work.

A. *Application Monitor*

In conjunction with kernel I/O monitoring, the application monitor captures all interactions between the user and the applications on the system. The monitor records 1) what UI element (e.g., a button) the user accessed and 2) the mouse or keyboard action that actuated that element.

The application monitor uses existing automation frameworks integrated into popular UI libraries and toolkits to gain visibility across applications without per-application customizations or source code access. Microsoft's .NET framework includes the Windows Presentation Foundation (WPF), which automatically includes support for accessibility. A recent survey of 41,760 software developer about the "most used non-web framework" found 37.1% selected .NET and 35.26% selected the .NET core [18]. Accordingly, a substantial portion of development is occurring in a framework that supports our automation technique. These tools provide an API into each application during compilation that allows external applications to query UI elements and receive notifications when the interface changes. Given the dominant market share of Windows and the popularity of the .NET platform and WPF, we focus on this platform and technology but recognize the same principles would apply to other similar tools. With just the that library's accessibility support, we can instrument elements of many popular applications, including Microsoft Office products, Google Chrome, and Mozilla Firefox.

Our application monitor leverages two mechanisms for collecting UI activity: event handlers and function hooking. The application monitor registers a global event handler for

focus changes, which includes any event that changes which UI element has the current focus for actuation, that occur in the user interface. This event handler runs asynchronously with the application receiving the focus change events, so it does not interfere with the user experience. However, because the handler is asynchronous, it is possible the handler will not finish execution before the network connection is initiated, making it more challenging for us to link the UI interaction to the network flow it precipitated. To address this, the application monitor additionally uses the Microsoft Windows `SetWindowsHookEx` function to hook all mouse clicks and keystrokes (represented by the lines from the mouse and keyboard to the application monitor in Figure 3). We record the element activated by each mouse click before the event is actually delivered to the recipient application.

Any time an event handler or hook function activates, the application monitor records which elements in the UI were accessed and how they were displayed in the application. In each case, the monitor uses the UIAutomation library to obtain the widget, or UI element, that was activated or focused on by the event (represented by lines ① and ② in Figure 3). The UIAutomation library represents the UI as a tree, providing information about the element's ancestors in the tree and a route from the activate element up to the root tree node, which is the Desktop element in Windows.

A final function in the application monitor queries for the most recent UI actions associated with a process. Each time an element is activated, we store its record in a global vector. When requested by the network monitor (line ⑤ in Figure 3), the application monitor service iterates through the vector, from the most recent event to the oldest, searching for entries associated with the requested process ID of the application. It then encodes this information into a text string for transmission to the network monitor (line ⑥ in Figure 3). To avoid redundancy, if two or more UI elements share ancestors, we denote the shared ancestor and omit the common ancestors above it. The application monitor reports up to three records.

We use kernel drivers to verify user inputs and the administrative service can verify UIAutomation is loaded and no untrusted DLLs are present. This allows HARBINGER to differentiate between software and hardware inputs.

B. Network Monitor

The network monitor serves two primary functions. First, it intercepts all new network connections. Second, it coordinates the application monitor and logging system. The network monitor is implemented as a kernel-mode Windows driver. The driver uses the Windows Application Layer Enforcement (ALE) portion of the Windows Filtering Platform (WFP) to monitor all socket operations, including the creation of TCP and UDP connections (lines ③ and ④ in Figure 3). The ALE module monitors traffic at a per-flow granularity.

When the network monitor identifies a new network flow for a process, it queries the application monitor for the UI interactions that preceded the new flow. Specifically, the network monitor uses the inverted call model to send a message to the application monitor (lines ⑤ and ⑥ in Fig. 3). This call includes the process ID of the application making the network call, which is provided by the ALE classifier functions. The network monitor then packages the contextual information, along with the packet that created the new flow, and transmits the data to the logging system and awaits acknowledgement (lines ⑦ and ⑧). While awaiting a response, the driver queues the packets in the flow. Upon acknowledgement (lines ⑨ and ⑩), the driver marks the flow as approved in the WFP framework and re-injects all locally queued packets for transmission to the destination (lines ⑪, ⑫, and ⑬).

The network monitor communicates with a centralized network controller that manages the logging system. We use a subset of the OpenFlow 1.0 protocol between the network monitor and policy engine to encourage use in the software-defined networking (SDN) community. Prior SDN research and deployments show that OpenFlow's latency is acceptable for production use. We fit the elevated packet and context within a 1500 byte MTU to avoid packet division.

C. Network Monitor Module for Web Traffic

Web traffic is unique in that a remote server provides a browser with instructions on new network connections to form without requiring user involvement. To support this dynamic, we extend the network monitor to add a module that intercepts all packets on common HTTP and HTTPS ports to enable detailed analysis for these protocols. An ideal implementation of this module would extract the session key logged by the browser, decrypt the traffic in the kernel, parse the payload for URLs, and record those entries. Given the required engineering, our proof-of-concept uses a third-party tool.

In our approach, the network module holds the packet and sends a copy to our administrative service for processing. Within the administrator user account, we use the `tshark` tool to decrypt and decompress the HTTP/HTTPS traffic. We then examine the output from `tshark` to extract links and the IP addresses and hosts inside the URLs. We combine this data with the originally requested URL provided by the Application Monitor, allowing us to associate the original request with the subsequent traffic from the browser to any dependent or linked resources. This approach allows us to log what traffic is specifically associated with user interactions.

We store the decrypted links and associated host names and IP addresses in a local database for the browser application. This allows us to fully process the packet within the context of HARBINGERand dynamically determine the user-driven page load's content. Importantly, this only associates access to destinations specified by the remote web server. This would not allow malware or malicious extensions in the browser to simply introduce URLs to the browser's DOM locally to have them associated with a user action. Further, if malware prevents the proper session key from reaching the administrative service, it will effectively block the association of legitimate destinations in the system rather than creating false associations that could mask malware activity.

V. How can UI Data Help in Practice?

User interaction context may help analysts in myriad ways. While we cannot anticipate them all, we explore a particular application, network profiling, to demonstrate the concept. We explore methods to handle automated background activity and user initiated traffic. We show that UI information can be linked to network actions, demonstrate such links in practice with real programs, and illustrate the context's utility.

Our experiments are completed in a lab study without human subjects. However, a study with human subjects would require approval from an Institutional Review Board (IRB) and user privacy and risks would be a key IRB consideration. To scope this work, we defer a full analysis of privacy risks and controls. Instead, we note the HARBINGER tool could gather information and apply policies locally and redact UI sensor data when reporting to an SDN controller. Such techniques may enable more powerful policies while reducing the privacy risks to those of traditional network profiling.

A. Network Profiling: Accuracy, Precision, and the UI's Role

Application developers frequently use concurrency in user-facing applications. Parallel threads of execution will be dedicated to different tasks, such as checking for updates, rendering output, or waiting for end-user input. Some of these tasks may use the network. When performing detailed network profiling, the goal is to accurately (i.e., correctly) and precisely (i.e., exactly) relate network activity to the associated application task. While simple tools can give accurate but imprecise results, such as indicating traffic came from a given system, both high precision and accuracy are needed to distinguish user and automated behavior.

Network connections in response to user actions can be more precisely described as actions taken with 1) a *fixed remote destination* or 2) an *end-user influenced* destination. For example, a user clicking a "refresh" button in a weather application implicitly requests a network interaction, but delegates server addressing to the program (and to the entity that developed or configured the software). In contrast, a user's URL specifies the protocol and destination to use.

B. Distinguishing Atypical Communication in Microsoft Office

The Microsoft Corporation announced over 1.2 billion people use its Microsoft Office suite[1]. Given its popularity, we explore how well different sensor types can profile traffic.

In our testing, we create a set of workflows in Microsoft Excel, Microsoft OneNote, Microsoft PowerPoint, and Microsoft Word. We initially start these programs and leave them idle, classifying any network connections that result during the *idle period* as automated background traffic. We then manually created workflows, some of which trigger network traffic while others do not. We encode these workflows into scripts using the Sikuli framework [19]. While our kernel drivers would detect the Sikuli actions as software-originated rather than hardware originated, we deactivate them for this study to mimic actual human use. Sikuli lets us direct the software to enter mouse clicks and keystrokes to interact with specific UI elements. These workflows use fixed remote destinations.

TABLE I
Accuracy of sensors with fixed remote destinations with no malicious traffic (i.e., all inaccuracy is a false positive)

Application	Idle Period Samples	Training Samples	Testing Samples	Sensor Accuracy IP	DNS	UI
Excel	4,173	3,425	2,490	99.0%	100.0%	99.9%
OneNote	1,764	1,690	1,645	99.8%	100.0%	100.0%
PowerPoint	2,366	792	1,671	94.1%	99.9%	99.9%
Word	3,826	2,809	2,095	96.5%	100.0%	99.8%

We collect workflow UI and network activity during an initial *training phase* in which the Sikuli workflows run. We then later repeat those same workflows during a second *testing phase* to see if the sensors could recognize the behaviors. We used three sensors with varying degrees of precision:

1) **An IP header sensor**, which reports a match if a subset of IP header fields (i.e., a match rule of (IP_{dest} AND transport protocol AND $port_{dest}$)) appeared together *in either* the idle period or the training phase,
2) **A DNS-aware sensor**, relaxes the above matching by allowing different IP addresses corresponding to the same host name to result in a match (i.e., a match rule of (($host_{dest}$ OR IP_{dest}) AND transport protocol AND $port_{dest}$)) that appears *in either* the idle period or the training data, and
3) **A UI-aware sensor**, which reports a match if either:
    a) the DNS match rule appears *strictly in* the *idle* period (i.e., background traffic), or
    b) the DNS match rule appears in the *training* period with UI context in common (i.e., the same catalyst user action must occur)[2], or
    c) the IP address or host name of the destination appears in the UI context data (e.g., in a URL).

In Table I, we show the accuracy of the different sensors across the four applications. We see that the IP sensor was 94.1% to 99.8% accurate, the DNS sensor was 99.9% to 100.0% accurate, and the UI sensor was 99.8% to 100.0% accurate. The IP sensor was the least accurate largely due to IP load balancing: the remote server had the same DNS host name, but used different IP addresses than those observed in the training and idle phases. The DNS sensor was the most accurate, since it could accommodate IP load balancing, with the fewest requirements (i.e., least precision). The UI-aware system was almost as accurate as the DNS sensor, but failed to match 6 connections. We determined that some background traffic did not appear in the idle period, but did occur during the training phase, causing it to be misidentified by the UI-aware system but accurate in the less precise DNS sensor.

To gauge each sensor's sensitivity, we introduced anomalous traffic. We use the dynamically-linked library (DLL) injection technique, which is commonly used by malicious attackers, to

[1] https://www.zdnet.com/article/microsoft-by-the-numbers-2015-700k-windows-store-apps-1-2bn-office-users/

[2] We also cache IP addresses associated with host names to combat DNS pinning effects, but abstract those details to simplify the explanation.

TABLE II
ACCURACY OF SENSORS WITH FAUX-MALICIOUS DLL TRAFFIC
(ACCURACY DIFFERENCE FROM TABLE I IS FROM FALSE NEGATIVES).

Application	Idle Period Samples	Training Samples	Testing Samples	Sensor Accuracy		
				IP	DNS	UI
Excel	4,173	3,425	3,446	71.5%	72.2%	99.4%
OneNote	1,764	1,690	2,439	67.3%	67.4%	100.0%
PowerPoint	2,366	792	2,392	65.8%	69.8%	99.2%
Word	3,826	2,809	2,655	76.2%	78.9%	99.1%

create threads that randomly connect to servers observed during the idle and training periods. This replicates the scenario in which advanced malware burrows in legitimate software and attempts to propagate among an organization's systems. The injected thread runs within each of the tested Office applications and waits a random interval, averaging around 15 seconds, between network connection attempts. The thread logs its activities as ground-truth data.

In Table II, we show the accuracy of the different sensors across the four applications when this faux-malicious traffic is present. We see that the IP sensor was 65.8% to 76.2% accurate, the DNS sensor was 67.4% to 78.9% accurate, and the UI sensor was 99.1% to 100.0% accurate. Each of the sensors retained their false positives rates from before, but both the IP and DNS sensors failed to detect the faux-malicious traffic. The UI-aware sensor did better, but it missed 55 faux-malicious connections simply because the thread's timing aligned with Sikuli's interaction with a UI element that allowed such traffic in the training phase. Naturally, the attacker DLL could evolve to monitor user behavior similar to HARBINGERand time attempts to match the UI behavior. However, other techniques focus on detecting manipulation such as DLL injection and future stateful, connection-counting list enforcement systems could detect anomalies.

Finally, we explore the impact of user-supplied destinations on the sensors. We modify the Sikuli workflows so that the emulated user randomly selects a URL from a database of the top half-million Quantcast websites [20], types that URL (which is automatically transformed into a hyperlink by these applications), and then activates the hyperlink. Sikuli activated 240–420 URLs per application in the training phase and 149–390 URLs during the testing phase.

In Table III, we show the accuracy of the different sensors across the four applications when the emulated user proceeds to the user-supplied destination. The accuracy of the IP and DNS sensors decreases substantially for Excel, PowerPoint, and Word (between 61.7% and 70.4%) simply because the sensors misclassify user-supplied destinations (since the training data did not include all the URLs selected by the emulated user). The UI sensor recognizes the URLs and correctly classifies them in those applications, resulting in 99.8% to 100.0% accuracy. The accuracy of the IP and DNS sensors would have been worse if not for the fact that the applications pair each network request with a connection to a Microsoft service that appeared in the training data. The OneNote results differed because it delegated URL retrieval to a browser.

TABLE III
ACCURACY OF SENSORS WITH URL ENDPOINTS.

Application	Idle Period Samples	Training Samples	Testing Samples	Sensor Accuracy		
				IP	DNS	UI
Excel	5,419	3,733	1,958	69.3%	70.4%	99.8%
OneNote	1,913	1,929	1,025	97.8%	99.8%	99.8%
PowerPoint	4,427	1,249	1,652	62.7%	63.6%	99.8%
Word	6,002	4,273	1,486	61.7%	62.2%	100.0%

C. Exploring Web Browser Traffic

Web browsers and encryption pose an additional challenge because the content of a given website cannot be predicted in advance. To determine whether our approach is effective at linking user browser activity to network flows with dynamic content, we use a custom-built extension for the Chrome web browser that can extract all links from a web page. This extension runs after the web page is fully loaded, and Javascript has executed, allowing us to obtain URLs that are generated by executing client-side code. We consider this data to be the "ground truth" of the links a user could access when visiting a page and compare it with HARBINGER's server list.

We collect our data by running a Chrome web extension that allows us to load web pages sequentially. We download the top seven most popular websites from the Alexa ranking website [21]. Each time the Chrome browser finishes loading a target page and its dependencies, the extension then retrieves the next URL listed in the database. During this experiment, we run the link extraction Chrome extension and HARBINGER's web browser module to obtain the packet data and write it to a packet capture file. After retrieving each page, we use the `tshark` tool and our extracted TLS symmetric key database to decrypt and decompress each logged packet and extract the links[3]. We then compared the ground truth and HARBINGER's data set to determine classification accuracy.

We found the seven page loads resulted in 407 host names and IP addresses within those links in our ground truth data set. Those same page loads resulted in 406 host names and IP addresses in HARBINGER, a match of 99.5%. Accordingly, the websites would have only rare misclassifications.

D. Automatically Profiling Applications

While our analysis in Section V-B shows the impact of specific scenarios in four popular applications, we now explore whether the approach generalizes to other software and other workflows. In particular, we explore whether UI action can be reliably associated with network traffic. To do so, construct a "guided UI fuzzer" tool that essentially traverses each node in an application's UI tree, activating elements (e.g., buttons, menus items) to explore the tree. We add basic form field capabilities (e.g., entering URLs in text boxes next to labels containing "URL") and heuristics to avoid unfruitful paths (e.g., file selection dialog boxes in workflows such as opening or saving files). As in Section V-B, we treat the script's actions as if they were created by end user behavior.

[3]Section VI-A shows this can be done in real-time, but we separate the data collection and analysis here for simplicity.

TABLE IV
NETWORK CONNECTIONS REVEALED WITH OUR UI FUZZER TOOL.

Application	Unique Fuzzer Items Explored		Policies Generated			
				Back-	UI Dest. Type	
	UI	Flows	Total	ground	Fixed	Dyn.
Audible	304	19,092	69	45	24	0
Bing Dictionary	145	989	17	16	1	0
FileZilla	343	0	0	0	0	0
Notepad++	307	0	0	0	0	0
PowerPoint	96	3,899	37	34	3	0
Putty	743	4	1	0	0	1
VLC	96	20	2	2	0	0
Word	189	5,410	31	28	3	0

While running the HARBINGER system, we use our fuzzer on a set of 8 applications, as shown in Table IV, two of which overlap with Section V-B. We explore at least 96 unique UI elements in each program, resulting in 0-19,092 network flows in each. We generate policies from these, distinguishing background and UI-based flows. In one case, for Putty's connection window, the policy identifies a user-supplied field (the host name) as correlated with the network connection. In three other programs, FileZilla, Notepad++, and VLC, the fuzzer's UI data reveals one or more element likely to produce a network flow, but without actuating it. This happens in cases where the fuzzer does not know how to supply data for a field or due to the UI sensor privacy controls (e.g., avoiding certain editable field types), such as the video stream source for VLC. We note a policy could be created to cover such behavior.

In Audible, the fuzzer plays a variety of audio books. While the policies largely shared UI data, variances in titles produce different policies. A regular expression wildcarding the title can reduce the policy count while retaining effectiveness. The fuzzing process also reveals some application nuances. Both Notepad++ and FileZilla offload network activity to a separate process. While fuzzer enhancements may improve data collection, some manual review may still be required.

We find this fuzzing can automatically explore the UI. However, deployers could simply use a prolonged training phase to collect and analyze organizational data.

E. Summary of Network Profiling Results

The UI sensor's precision can reveal abnormal traffic while improving accuracy in traffic to end-user influenced destinations (e.g., via URLs). The technique works across popular applications and web browsers (even with TLS encryption).

VI. PERFORMANCE EVALUATION

We examine the end-to-end latency of classifying network flows in HARBINGER. The median delay is less than 3ms and it was less than 6ms for 95% of trials.

The Application Layer Enforcement (ALE) layer in Microsoft's Windows Filtering Platform (WFP) supports efficient flow classification decisions by exposing a kernel-mode interface for drivers. ALE intercepts all packets in a flow until the flow is approved. Once approved, all subsequent packets in a flow are approved and bypass the flow classifier. HARBINGERelevates each packet in the flow to a network controller's policy engine for a decision and then locally enforces filtering decisions in the host's ALE classifier. With ALE, the only overheads are associated with the first packet in the flow, which triggers the query to the policy engine. This behavior is similar to reactive flow processing in the OpenFlow protocol.

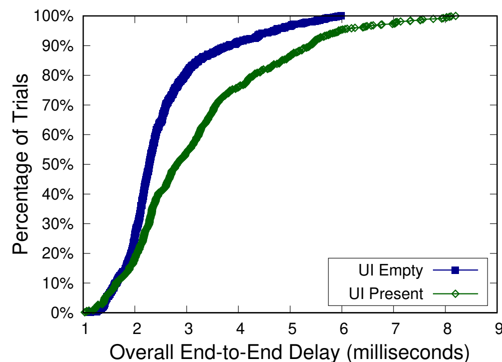

Fig. 4. CDF of end-to-end delay introduced by HARBINGERon new flows. There were 765 traces (with 34 outliers removed) that did not have UI output present ("UI Empty") and there were 743 traces (with 43 outliers removed) that the UI information populated ("UI Present").

We first examine the end-to-end delay associated with processing new flows in HARBINGER. We used two virtual machines running on separate physical VM hosting servers. The policy engine and controller ran in an Ubuntu 16.04 VM that was allocated 1 core and 2GBytes of RAM on a 12-core 2.6GHz VM server with 64 GBytes of RAM. The Windows 7 end-user machine ran in a VM allocated 2 cores and 4 GBytes of RAM on a 20-core 3.1GHz VM server with 128 GBytes of RAM. The two VM servers were connected via a 1 Gbps Ethernet connection to the same Ethernet switch on our organization's production network. We did not see noticeable changes in CPU or memory usage on these systems. Our tests explored Google Chrome, Internet Explorer, Microsoft Excel, Microsoft OneNote, Microsoft PowerPoint, Microsoft Word, Microsoft Visual Studio, Mozilla Firefox, PyCharm (a software IDE), Slack, and Skype.

We use the `KeQueryPerformanceCounter` kernel-mode routine to measure end-to-end delay with a 0.1 microsecond (μs) resolution. We measure from when the first packet of a flow reaches HARBINGER's network monitor to the time when HARBINGERdenies the flow or re-injects the packet(s).

In Figure 4, we plot the cumulative distribution function (CDF) of the results of these experiments. The end-to-end delay for both cases where a UI was present and when it was empty for the majority of connections is less than 3 milliseconds. All the traces with an empty UI trace (excluding outliers) completed in less than 6 milliseconds while 95% of the traces with a UI finished in less than 6 milliseconds.

To measure the computation requirements at the network controller and policy engine, we measured the processing time across a total of 653,189 trials and 31 policies. The average controller running time was roughly 835 μs with the biggest contributing components being the encryption of the response (290 μs), decryption of the query (159 μs), the parsing of the UI context (117 μs), and the evaluation of the flow verses the existing policy rules (76 μs). The network controller and policy

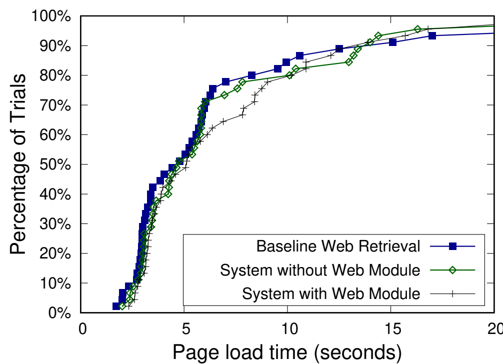

Fig. 5. CDF for page load times, 45 trials in each data set

engine can sequentially process thousands of flows a second. In a subsequent experiment, we ran a controller in a VM on our 20-core 3.1GHz VM server and found that each each core allocated to the VM was able to process between 1,350 and 1,500 new flow requests per second. A four-core VM was able to handle around 5,450 new flows per second without building a queue. With parallelism, a multi-core controller may service tens of thousands of new flows per second.

A. Web Traffic Module Performance

As mentioned in Section IV-C, our proof-of-concept implementation includes inter-process communication, kernel/user mode transitions, and other overheads that would not be present in kernel mode implementation. Our results are thus an upper bound for the web traffic performance costs.

We use a script to launch the Chrome browser in incognito mode to avoid caching effects. The script specifies a page that Chrome should load. We obtain an overall page load time using a Chrome extension that measures the amount of time to load each web page and its dependencies. We measure the amount of time required to load a set of pages 1) without HARBINGER running, 2) with HARBINGERbut without the web browser module, and 3) with our complete system.

From our experiments on 45 page loads selected from the Alexa top 45 web sites, we found that the median page load time was around 4.5 seconds when HARBINGER was not loaded. The median page load time was around 4.6 seconds when HARBINGER was loaded without the web module, which includes the time required to seek approval from the SDN controller. When the HARBINGER web module was loaded, this increased to around 5.1 seconds. We show the cumulative distribution functions of these page load times across these three experiments in Figure 5. These results show that the web module adds about 10% of the page load time for many page loads. This delay may affect the user experience in some instances, but may not affect other applications. With additional engineering efforts, we expect that time to decrease.

VII. CONCLUDING REMARKS

We show that UI sensors can help analysts in tasks such as network profiling. Those sensors can monitor UI interactions efficiently and universally without significant application-specific instrumentation or overhead. It was deployed across a variety of desktop applications, including a web browser using TLS. The sensors have accuracy of at least 99.1% in experiments and is sensitive enough to detect anomalous and malicious traffic. Such UI context may further help analysts with other tasks, such as reconstructing errors or auditing.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1422180. Shue holds stock in ContexSure Networks, Inc., an arrangement that has been reviewed and approved by WPI.

REFERENCES

[1] K. Suneetha and R. Krishnamoorthi, "Identifying user behavior by analyzing web server access log file," *IJCSNS Journal of Computer Science and Network Security*, vol. 9, no. 4, pp. 327–332, 2009.
[2] 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*, 2010.
[3] 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.
[4] N. Provos, "Improving host security with system call policies," in *USENIX Security Symposium*, Berkeley, CA, USA, 2003, p. 18.
[5] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *International Conference on Software Engineering*, 2014, pp. 468–479.
[6] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, 2016.
[7] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *ACM Conference on Computer and Communications Security*, 2017, pp. 1285–1298.
[8] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *USENIX NSDI*, 2014.
[9] W. Cui, R. Katz, and W. Tan, "BINDER: An extrusion-based break-in detector for personal computers," in *USENIX ATC*, 2005.
[10] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM CCS*, 2002.
[11] J. Kwon, J. Lee, and H. Lee, "Hidden bot detection by tracing non-human generated traffic at the zombie host," in *International Conference on Information Security Practice and Experience*, 2011.
[12] G. A. Fink, V. Duggirala, R. Correa, and C. North, "Bridging the host-network divide: Survey, taxonomy, and solution," in *USENIX Conference on Large Installation System Administration*, 2006, pp. 20–20.
[13] Y. Jang, S. P. Chung, B. D. Payne, and W. Lee, "Gyrus: A framework for user-intent monitoring of text-based networked applications," in *Network and Distributed System Security Symposium (NDSS)*, 2014.
[14] W. N. Bhukya, S. K. Kommuru, and A. Negi, "Masquerade detection based upon GUI user profiling in Linux systems," in *Annual Asian Computing Science Conference*, 2007.
[15] M. Stiegler, A. Karp, K. Yee, T. Close, and M. Miller, "Polaris: virus-safe computing for windows XP," *Communications of the ACM*, 2006.
[16] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *IEEE Security and Privacy*, 2012.
[17] J. Shirley and D. Evans, "The user is not the enemy: Fighting malware by tracking user intentions," in *New Security Paradigms Workshop*, 2008.
[18] D. Ramel, "Stack overflow: Old .net framework usage still beats 'most loved' .net core/.net 5," https://visualstudiomagazine.com/articles/2021/08/03/so-survey-2021.aspx. Last accessed Oct. 18, 2021.
[19] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *ACM Symposium on User Interface Software and Technology*, 2009, pp. 183–192.
[20] Quantcast, "Top websites," https://www.quantcast.com/top-sites/. Last accessed Sept. 30, 2018.
[21] Alexa Internet, Inc., "Top sites," https://www.alexa.com/topsites. Last accessed Oct. 18, 2021.