UTrack: Enterprise User Tracking Based on OS-Level Audit Logs

Yue Li
College of William and Mary
yli20@email.wm.edu

Zhenyu Wu
Google Inc.
zhenyu.5wu@gmail.com

Haining Wang
Virginia Tech
hnw@vt.edu

Kun Sun
George Mason University
ksun3@gmu.edu

Zhichun Li
Stellar Cyber
zhichun.li@gmail.com

Kangkook Jee
University of Texas at Dallas
kangkook.jee@utdallas.edu

Junghwan Rhee
University of Central Oklahoma
jrhee2@uco.edu

Haifeng Chen
NEC Laboratories America
haifeng@nec-labs.com

ABSTRACT

Tracking user activities inside an enterprise network has been a fundamental building block for today's security infrastructure, as it provides accurate user profiling and helps security auditors to make informed decisions based on the derived insights from the abundant log data. Towards more accurate user tracking, we propose a novel paradigm named UTrack by leveraging rich system-level audit logs. From a holistic perspective, we bridge the semantic gap between user accounts and real users, tracking a real user's activities across different user accounts and different network hosts based on causal relationship among processes. To achieve better scalability and a more salient view, we apply a variety of data reduction and compression techniques to process the large amount of data. We implement UTrack in a real enterprise environment consisting of 111 hosts, which generate more than 4 billion events in total during the experiment time of one month. Through our evaluation, we demonstrate that UTrack is able to accurately identify the events that are relevant to user activities. Our data reduction and compression modules largely reduce the output data size, producing a both accurate and salient overview on a user session profile.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**.

KEYWORDS

Audit Logs; Forensics Analysis; User Tracking

ACM Reference Format:
Yue Li, Zhenyu Wu, Haining Wang, Kun Sun, Zhichun Li, Kangkook Jee, Junghwan Rhee, and Haifeng Chen. 2021. UTrack: Enterprise User Tracking Based on OS-Level Audit Logs. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422337.3447831

1 INTRODUCTION

Nowadays, cyber-attacks have been becoming more sophisticated and stealthy. In an Advanced Persistent Threat (APT) attack, an attacker may lurk in the target network for more than half a year on average, escalating and maintaining the access privilege without being caught [40]. As a result, there is an increasing demand of user tracking inside an enterprise network, in order to improve the visibility for the network monitoring, and help security analysts to make informed decisions on the detection of insider attacks and targeted APT attacks. A recently enabled paradigm in the security industry, called User Behavior Analytics (UBA) [38, 39], is built upon this foundation. UBA categorizes a range of techniques that keep monitoring user activities and identifying those that deviate from normal user sessions. While UBA is a rather broad concept that can be applied to many scenarios on a different level, granularity, and scope, its fundamental building block is to accurately identify and model user activities. Capturing user activities with an inaccurate or incomplete view could result in incorrect detection or analysis.

Towards more accurate user modeling and verification, contemporary UBA approaches attempt to fuse data from different data sources for creating a more comprehensive risk profile [33, 41]. Though they are useful in many scenarios [33, 39, 41], an inherent limitation is that they all lack a holistic view on systems since data are collected from only a couple of security-sensitive applications, such as firewalls and proxies. Under such a setting, many meaningful events could be missed, not to mention the difficulties of correlating data with different syntax and semantics from a variety of sources. A natural approach would be to leverage log data at the operating system (OS) level, which can record data for all applications under homogeneous syntax and comprehensible semantics. Such an audit log system is widely deployed in many security infrastructures [22–24, 26, 27], mainly for forensics purposes. SLEUTH [18] and HOLMES [29] leverage system logs to identify APT attacks based on abstracted security sensitive activities: "tags" for SLEUTH and "Tactics, Techniques and Procedures" (TTPs) for HOLEMES.

In this paper, we present a novel user tracking system, named as UTrack, by leveraging the rich system log data to universally monitor user session activities. In addition to focusing on identifying, consolidating, and scrutinizing security sensitive events [18, 29], UTrack is *user-centric*. UTrack does not pay more attention on pre-defined "sensitive" read or write. Instead, the goal of UTrack is to present a user's activity profile accurately and concisely, such that more domain-specific behavior can be audited. For example, an employee copying a large amount of digital assets from the company should be known by UTrack.

We identify and tackle two major challenges. The first is to bridge the semantic gap between user accounts and human users in both *in-host* and *cross-host* scenarios. This is done by tracking causal relationship among processes through the user session root and correlating network events to identify network control channels. The second challenge is to address the "needle in a haystack" problem stemmed from the huge volume of log data through a variety of data reduction techniques. Unlike many previous works on log data reduction [25, 46] that aim at information-lossless reduction, our data pruning approach is to prune data that may carry meaningful information but are out of the scope of user activity tracking.

We deploy UTrack in an enterprise network that comprises more than 100 hosts running either Windows or Linux operating systems with real users. The users are well aware of the setup. This is a general setup among many enterprises that the company devices are actively monitored. We manage to process log data from all the hosts on a single machine, and demonstrate that UTrack is able to accurately identify and concisely present the events that represent activities of a real user inside the network in a human-consumable fashion.

In summary, we make the following contributions.

(1) We develop a new universal user tracking mechanism (UTrack) based on OS-level audit logs. UTrack aims to *bridge the semantic gap between human users and computer user accounts* by identifying and associating system events that appear in different user accounts and different hosts but belong to a single user session.
(2) We apply effective *data reduction methods on user session profiles* to achieve a scalable and salient presentation. The reduction mainly involves detecting interactive processes and modeling common data patterns.
(3) We implement UTrack in a real enterprise environment, with data collected from more than 100 hosts. Our evaluation results show that UTrack is accurate and concise in presenting user activities. UTrack scales well with a low resource consumption.

The rest of the paper is organized as follows. Section 2 describes the motivation and challenges of an OS-level log based universal user tracking system. Section 3 presents system overview. Section 4 elaborates how UTrack tracks user sessions across different accounts and different hosts. Section 5 presents various techniques UTrack adopts to pinpoint relevant events. Section 6 details the implementation and evaluation of UTrack, and Section 7 discusses more use cases. Section 8 surveys related works, and finally, Section 9 concludes the paper.

2 MOTIVATIONS AND CHALLENGES

2.1 Motivations

Contemporary user behavior monitoring is mostly done on disparate applications and services. However, such a methodology has multiple drawbacks that limit the usability of the monitoring system. The first drawback is the lack of completeness. In these systems, only a small portion of user activities are recorded and analyzed, since the logs are only generated from applications that are usually perceived to be of strong security indication, for instance, a firewall, a web proxy, or a sensitive database service. All other user activities are not actively monitored. However, a successful attack, especially an APT attack, usually comprises many individual steps. The traces of each step may be buried in seemingly less interesting events that are not recorded by applications. By connecting these dots, one may detect an intrusion that cannot be identified by conventional user behavior analytics. In contemporary user tracking schemes, the auditor lacks this holistic view on the entire system.

The other limitation is the difficulty of correlating log data. Data collected from different services and applications may be of different formats, granularity, and semantic levels. Parsing and correlating data from different sources is very challenging. As a result, data from individual sources are independently handled and analyzed in many cases. Shashanka et al. [33] attempted to associate subjects from different data sources, such as different IP addresses and user accounts. However, the capability of such an association is limited to a small scope, where the subjects are tightly bounded. Therefore, the inspector lacks view on the connections among critical pieces of puzzle from all data sources.

System Opportunities: A universal user activity tracking system, which monitors activities of all users inside an enterprise network, is very useful to resolve or mitigate the aforementioned problems. However, recording all activities of individual users in the entire network may incur significant system overhead. To balance the trade-off between system overhead and data granularity, we leverage an OS level log system to collect data from each host inside a network. The OS level log system collects low level system objects, such as processes, files, and network connections, which largely preserve the running states of a computer at a certain time. Thus, it can be used to accurately reconstruct the causality among objects with clean semantics. Meanwhile, the data volume is at a manageable level. Nowadays, many enterprises have deployed such a log system for forensics purposes [22–24, 26, 27].

2.2 Challenges

2.2.1 Accurate Modeling of User Behaviors. When processing audit logs, a user account is often considered equivalent to the user itself. This is mostly true in some high-level applications, such as Facebook and Twitter. However, the assumption no longer holds when it comes to low-level OS events.

Unlike application-specific logging that is clearly defined and has much higher semantic awareness, a generic OS-level log system monitors events with respect to individual user accounts. In an enterprise network, a user may have multiple user accounts, and a user account could be accessible by multiple users. For instance, a network administrator could access both its personal account and the root account on a web server. The web server may also

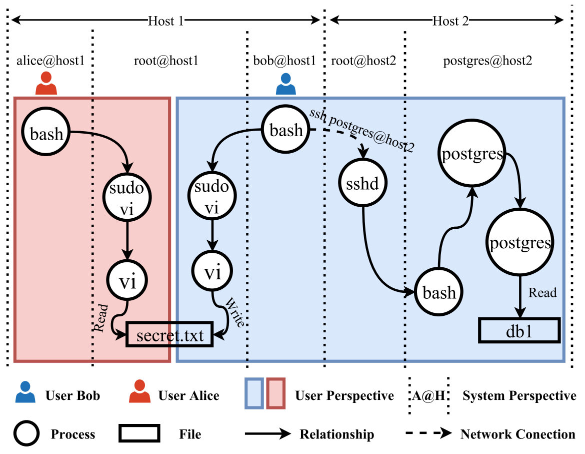

Figure 1: Users and User Accounts – an Example

be managed by several system administrators. As observed in our network, the discrepancy mainly comes from the following three scenarios.

Account Transition Managing account privilege and ensuring proper isolation among different privilege levels are essential to an operating system. However, user accounts with lower privilege sometimes need higher privilege to accomplish certain tasks, which is mainly done by 2 ways: (1) setting the UID of a process (e.g., "ping" command) or (2) having a higher privilege account to do the task (e.g., through "sudo" or "su" commands). Thus, a simple task done by a single user may involve several user accounts, and the same account may also be involved in activities performed by different users. A more comprehensive example is shown in Figure 1. In this example, two users Alice and Bob access a same file "secret.txt." However, from the perspective of the operating system, the secret file is indeed accessed by two "vim" processes of the "root" account. Furthermore, whoever is granted the root privilege is able to interact with the system on any other account's behalf. Although this does not usually happen in normal operations, it is possible that an attacker exploits this system trait to mask its malicious behaviors.

System Service. In a typical operating system, there are many applications and services running in the background. Meanwhile, many system accounts are created in the sense of security groups to achieve a finer granularity of access control for these services. Although these accounts do not represent any individual user, they are delegated certain tasks by other users. Two examples, "sshd" and "postgres processes", are shown in Host 2 in Figure 1. In this case, the PostgreSQL database server receives a request of accessing the database, and in response, the server daemon creates a child process to process the request. All these activities at the database server are recorded as from the account "postgres", regardless of the real user that is in fact accessing the database.

Credential Sharing It is possible that a same account is shared among multiple real users. A typical scenario is the "root" account on a server, which may be managed by several developers or administrators. Therefore, it is important to find the real performer of an event, especially when multiple real users log into the system using the same account.

In general, there exists a semantic gap between user accounts and human users. Solely relying on the user accounts to track the behavior of a user is not reliable as it lacks proper linkage of user account transition and service account delegation. We realize this semantic gap, and discard this intuitive but invalid assumption in our user tracking system. To clearly set boundaries between the two concepts, thereafter, the term "user" always indicates a real human user, while the term "account" always indicates a user (or system) account in a computer system.

2.2.2 Identifying Data Triggered by Users. The other challenge we face is to sift out data that are directly related to user behaviors. We observe that only a small portion of the data are triggered by direct user interactions with the computer, and others are spontaneous or scheduled system events, such as automatic updater and cron jobs. However, human interactions are the natural target of a user tracking system. In light of this, we attempt to identify system events that are triggered by users' actual interactions, which achieves a much higher scalability and cuts off unnecessary distractions for security auditors. However, for cleaner data semantics and resource conservation, OS level log systems usually only record the causal relationships among primitive system objects, such as processes, files, and sockets. They do not usually keep track of the operations on I/O devices, such as a click on the mouse or a tap on the keyboard. Without such information, it becomes non-trivial to identify events stemmed from users' interactions. To achieve our goal, we follow the lineage of the UI management components to identify possible processes that have an open interface to the users and rely on many useful features to determine interactive processes.

We also need to address the semantic gap between the actual high-level user behaviors and low-level system interpretations. In many cases, a simple operation from a user may result in a large number of system events. Though these events are indeed triggered by the user's behavior, they carry little information as most of the steps are highly repetitive and predictable. To eliminate this redundancy, we try to model file sets that are frequently accessed by processes, and only record the events that do not fit in the model. In addition, we also model repetitive execution "branches" of a process and compress these repetitive ones.

3 SYSTEM OVERVIEW

Nowadays, many organizations have started to deploy an agent on each host in their enterprise networks. UTrack works under the same context of these forensics-purposed OS level log systems. Three types of system objects (process, files, and network sockets) and their interaction events (e.g., a process creating a child process or reading from a file) are recorded. Each event also carries attributes that describe the activity, such as the event time, user account, file pathnames, and socket IP addresses, etc.

UTrack aims to help a system auditor to understand the activities of users inside an inter-connected enterprise network by associating both in-host and cross-host activities performed by the same user in a specific user session. The input to UTrack is a data stream collected from all hosts, and it can be either a real-time stream or an offline history database. Generally, it consumes the data from a start time (T_s) to an end time (T_e), and outputs user session profiles to describe the activities of a user session within the time period. If

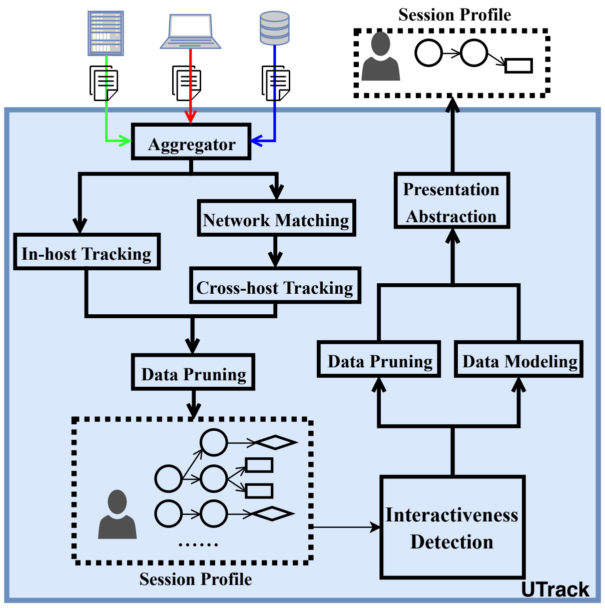

Figure 2: UTrack Overview

the data is an online stream, the end time T_e is set to a distant future time. The user session profile is represented in a forest structure, and the time is usually set in terms of weeks, days, or hours in different use cases.

Most forensics techniques consider a system object, such as an identified trojan process, as a Point of Interest (POI), and aim at understanding the provenance or impact of POIs. In contrast, UTrack attempts to understand the behavior of a user in a session. In other words, the user itself is the POI. In most cases, the behavior of a user is much more complex to describe than a single attack incidence in the system. Similarly or even worse, UTrack suffers from the same data explosion problem, which makes it difficult to focus on the real interesting events. Thus, it is imperative for UTrack to identify and keep the most relevant data, which can also help save system resource.

Figure 2 illustrates an overview of UTrack. UTrack consumes a data stream of log events from the aggregator, which receives, sorts, and sends out the data from the agent-enabled hosts. The first task of UTrack is to construct user session profiles by correlating both in-host and cross-host activities, mainly relying on the process lineage and network event matching. However, this session profile contains a large amount of system-generated data that may not be directly related to the user's operations, but can easily overwhelm other interesting events. To mitigate this issue, we apply interactiveness detection on the user session profile to identify the processes that have actual interaction with the user. This step directs us to the events that are more relevant to user tracking. We also model the files, network connections, and "sub-branches" of the interactive processes to further compress the low-entropy events. The output of this step is a more salient session profile, which can be directly construed by human auditors or be inputted to further security measures.

UTrack works in an online fashion. It gradually builds the user session profiles while consuming system events on the fly. It is important for a UBA system to promptly analyze the data, so that anomalies can be identified in their early stage and triaged to prevent further damages. On the other hand, UTrack can also work offline in forensics analysis by reconstructing the data stream from the log database. In order to facilitate this feature, we build a data replayer to replay the history data from the database, which is detailed in Section 6. This data replaying tool is also useful on implementing, debugging, and evaluating our user tracking system.

Note that UTrack does not aim to replace conventional forensics techniques, such as backtracking or forward tracking. Instead, it is indeed complementary to those techniques to better secure an enterprise network. Nowadays, it is a common case that people do not really make good use of big data, and a large amount of collected data remain in the warehouse without generating any useful insights. UTrack demonstrates a new perspective to better leverage the collected rich system data for system security and management purposes.

4 UTRACK EVENT ASSOCIATION

UTrack is capable of tracking users across an enterprise network by linking events from different hosts. Note that we no longer depend on the owner of the process (i.e. the user account) to determine the real performer of an event. Instead, the process owner is only used as side information to give a hint of who the performer might be. In the following, we first introduce the tracking mechanism on a single host and then extend it to the cross-host scenarios.

4.1 Tracking In-host User Activities

4.1.1 Process Lineage. Modern operating systems usually maintain all alive processes in a tree structure. For example, in Linux, every process, except the *init* process, is forked by a parent process. This parent-child relationship widely exists among processes and is useful to determine the performer of most system activities. Specifically, we consider the user of the child processes to be the same as that of the parent process, unless we have a special reason to cut the lineage and attribute the parent and children to different sessions. For instance, a user might open a Bash terminal and run the "ls" command in the terminal. Since the "ls" command is executed by a child process forked by the bash process, UTrack considers both processes to be performed by a single user.

This parent-child relationship is a fundamental building block of many forensics analysis techniques [22–24, 27], which usually expand the investigation from POI, i.e., the detected point of an attack. Timing is also considered to mitigate the possible dependency explosion and find the most relevant events. In contrast, UTrack tracks all the parent-child relationships among processes for constructing a more complete user session model.

We do not keep track of file data control flows, which are usually considered in forensics techniques. This is because when a process has written to a file, the file has a causal relationship with all the processes that read the file afterwards; however, this causal relationship is out of the scope of user tracking where the user activities are the target. Furthermore, it introduces too many dependencies that may unnecessarily complicate the analysis.

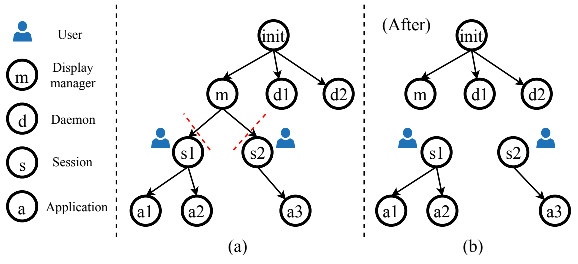

Figure 3: Session Root Isolation

4.1.2 User Log-on Sessions. Since one OS usually structures all its processes in a tree (or forest in Windows), when activities from multiple users are recorded in a host, it is far from adequate to solely rely on the process lineage for tracking each user. Thus, we must find a way to attribute related nodes to different users. A critical observation is that a user must have an interface to interact with the computer, and usually the first step is to log on the computer for user authentication. In addition, the OS usually organizes the processes under one user log-on session in a tree structure, and normally there is a root node as the ancestor of all processes created in the user session. We call this node a *session root*. Figure 3 shows an example of Linux instance. Each user logging in the system has a corresponding session root (i.e., node s_1 and node s_2), which is usually a child process of a running service. Their activities (for instance, open an application) are reflected in the subtrees under the session root. UTrack utilizes session roots to identify the activities of each user. It brings us two benefits. First, it helps to separate processes and activities triggered by users from those generated by the OS or system services. Second, it can differentiate activities among multiple users who have logged on the host simultaneously.

UTrack identifies session roots from several known patterns. For a normal user, the most common way to interact with the computer is via a Graphic User Interface (GUI). Even command line interactions are included since the terminal window itself is created in the desktop environment. For example, in Linux, the X display manager (a process usually named *dm) manages the login screen and organizes a user session in child processes. In our experimental environment, the most common display manager is *lightdm*, and thus the session root in this case is a lightdm session child with a session ID. When users log on a server through virtual consoles or no X server is available on the server, the session root is */sbin/login*, which is a child of the system *init* process. It is even easier for Windows, as it is a GUI-based OS and the user interaction with the OS is usually through the GUI. We determine the windows process “winlogon” as the session root, since it initiates the user authentication process and becomes the root of the desktop environment when the login succeeds.

Remote logins, such as through ssh and telnet, are envisioned as user cross-host activities since events from multiple hosts need to be correlated to track the relations. We elaborate how we handle cross-host activities in Section 4.2. When logins are from hosts that do not have an agent installed or the login happens before the tracking start time T_s, we identify session roots based on the service pattern. For example, the ssh daemon creates a dedicated shell for whoever has successfully logged on the computer via ssh. As such, the dedicated shell is considered as the session root.

4.2 Tracking Cross-host User Activities

In an enterprise network with many inter-connected hosts, one user may need to work on, or request resources and services from servers. It is critical to track the cross-host user activities in order to achieve a better coverage than local-only tracking. A number of previous works have been developed to help understand how a request is processed in a complex distributed system using middleware or application level instrumentation [5, 37], statistical inference [2, 31], or system call log and analysis techniques [32, 36]. However, they all cannot accurately work under generic-purposed OS level logs.

We propose to track cross-host user activities based on one key observation that after receiving remote requests, a server will act on behalf of the requester. Most servers have a daemon listening to incoming requests and processing the requests accordingly. There are two types of server architectures, namely, event-driven servers and worker-based servers. For the event-driven servers, since a thread could handle multiple incoming requests in a non-blocking, interleaving manner, it is hard to correlate a remote request with the corresponding activities of the server without specific assistance from the server. Therefore, our main focus is on the worker-based servers, where a network request is solely handled by a worker. Worker-based servers are popularly used in enterprise networks with a moderate number of users due to the ease of coding and maintenance. A worker-based server may support two working modes, namely, on-demand worker creation and a pre-allocated worker pool. As an example of the first mode, *sshd* daemon accepts a remote network connection, and creates an interactive command language interpreter process, such as a *bash* terminal. Thereafter, the newly created command interpreter process is controlled by the requester, and any activities performed by the process should be attributed to the requester, regardless of the user account that owns it on the server. We call this process a *delegate* of the remote user.

It is more tricky to handle the worker pool mode. In UTrack, one process node in a user session completely belongs to the user. However, it does not fit well with the mode of a worker pool, where multiple long-living workers are pre-created and each worker only dedicates a partial of its lifetime for a network request. In order to accommodate such a case, we introduce a new notation – *virtual process* – to model a span of the worker's lifetime. When a worker begins to work on a requester, we create a new node (i.e., the virtual process) in the user's model, and the new node records all the activities of the worker during this time. We illustrate this process in Figure 4, where two users make requests to a server at different times. The server dispatches the same worker to access two different files, f_1 and f_2. A virtual node is created to represent the time lapse that a worker is processing each request. Eventually, the user model is constructed with each user associated with its own virtual process, which carries all data during the time when the real worker handles the individual request.

To track cross-host user activities, the first step is to find the communication channel between the server application (i.e., responder) and the user-controlled application (i.e., requester). Next, we try

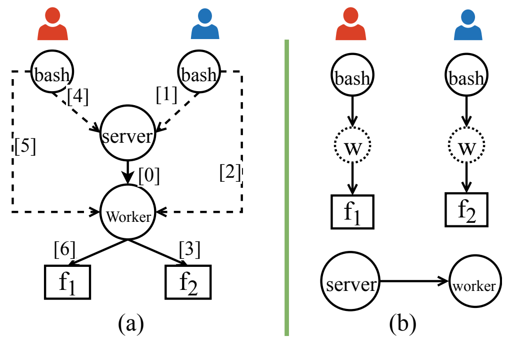

Figure 4: Virtual Process

to identify a worker for the request using a rule-based method. In both worker modes, we observe that after establishing a network channel as a connection acceptor, a child process or a sibling (when the listener and workers are siblings) of the server process immediately accesses the same network channel and generates a number of events. Based on this pattern, we can determine the worker and attribute all the activities of the worker to the remote requester.

4.3 System Cold Start

UTrack consumes data from agents during a pre-set time period to help understand the user sessions; however, it may encounter the cold start problem, namely, the history data is not available and the agents report events on scattered processes. If so, the linkage among processes may be missing. Since UTrack relies on the causal relations to identify user sessions, it requires to reconstruct these relations between processes. To address this problem, the agent periodically collects a system snapshot that stores the child-parent relationship among them. We use this information to reconstruct the causality relations among stand-alone processes and further extract user sessions in the reconstructed process tree. Note that the parent-child relationship recorded in the snapshot may not be coherent with that generated by UTrack, due to possible process delegation, user session identification and isolation, or the adoption of orphaned processes, etc. This discrepancy is in fact beneficial to our user tracking scheme. For instance, if a user starts a system service during the tracking period, the system service is regarded spawned by the user, and the activities of the service can be attributed to the user. However, if the tracking period is after the system start time and the service daemon is adopted by the "init" process, then the service becomes a part of the operating system and cannot represent any user. As such, we only reconstruct the parent-child relationship when the child process has no existing parent in the UTrack model.

4.4 Scope

UTrack aims to connect events that are cross-host and cross-accounts. However, there are cases where UTrack cannot handle. For instance, it could fail to identify causality among processes due to inability to track IPC mechanisms, such as shared memory and shared files. It also cannot handle the event-driven servers, like those run NGINX. Similar limitations can be found in previous works [6, 32, 36].

5 PINPOINTING USER ACTIVITIES

After correlating activities of users regardless of the process owner, UTrack collects user session profiles that keep track of all processes, files, and sockets in the memory during the entire tracking period. As a result, the generated data can become very large, and interesting events may be buried in piles of less-relevant data. Therefore, it is essential for UTrack to identify and keep only relevant and useful events. Redundant data must be pruned to release the pressure of huge system resource demand and to keep the security auditor from unnecessary distractions.

When conducting data pruning, we stick to the user-centric mentality by sifting out the events that are directly related to the user's interaction with the computer system. This is because that a user session profile contains a collection of processes that are only used to facilitate user or system operations. For example, Ubuntu provides a number of tools and services, such as GNOME Virtual File System(gvfs) for I/O abstraction, update-notifier for newer version checking, zeitgeist for logging user activities, etc., which are less relevant to user's actual behaviors. In contrast, interactive processes are the processes that a user interacts with, such as a Bash shell or UI-based programs like Firefox, Notepad, etc. The behavior of interactive processes is a genuine reflection of the user operations. However, it is a challenge to identify those interactive processes from our OS level logging information, which does not include any user actions, such as mouse clicking or keyboard input. UTrack relies on passive observation and prediction to find interactive processes, and multiple features have been identified to help distinguish interactive processes from other processes.

In addition to the interaction-oriented sifting, the data can be further compressed due to the highly repetitive patterns found in processes and files. We observe that the interactive processes are prone to generate sub-processes "branches" for different tasks. These branches could be similar to each other, regarding to the executable names, arguments, and files that are read. In many cases, these monotonous data can easily dominate a session profile and occupy over 90% data of the user profile. To address this issue, we model both the activities of an interactive process and the common files that are read by each executable, which significantly reduce the complexity of the user session profile.

5.1 Interactiveness Detection

The purpose of interactiveness detection is to find user-triggered events. We consider user-triggered events to be events directly resulting from a user action, such as opening a file using Notepad, etc. It should be noted that technically, all events are results of human user activities, since background procedures and processes, even the operating systems are installed by the user. However, since these processes are mostly regulated and expose behaviors dual to bots, we consider them to be non-user-triggered. Conceptually, we envision this procedure being similar to find bots/crawlers in a network, where the bots are essentially programmed by human users, but they expose very different behaviors and have little relation to active genuine users.

The interactiveness detection relies on passive observation of the OS events, so it faces several noteworthy challenges. First, passive observation is believed to be less accurate than active detection solutions, such as Catpcha [34, 44]. Second, we do not have a specially tailored log system as those used in bot detection [13, 45], or any side information such as social graph [7, 9, 13, 43]. Lastly, system level events are low-level data whose semantic meanings are harder to derive. Sometimes, we need to associate other related events to truly understand the actual user operations.

To categorize unknown processes, we develop a machine learning approach that uses a number of useful features to distinguish an interactive process from other processes. Note that we need to keep the actual activities that are usually represented by child processes of an interactive process. For example, an interactive shell may run many commands, which is executed transiently. These commands are not considered interactive processes. However, they represent the user's activities and should be studied.

An important and new feature we use is the entropy of activity batches. A fundamental observation is that the interactive processes have irregular activities, due to human involvement. Thus, a process performing tasks at a fixed time interval is unlikely to be controlled by a real human user. However, treating each individual event as a task is problematic since a single task usually constitutes many steps and events. As a solution, we preprocess all these events that are generated by the process to form a group of event batches, in which each batch represents a high-level task or operation. A batch consists of a group of events where each pair of adjacent events has an inter-arrival time of less than a threshold T. T should be carefully selected since it may result in leaving all the events into a single huge batch when it is too large, or losing the causality among events that are generated from a single task when it is too small.

We envision the time interval between two consecutive activity batches as a random process and decide if a random process is regular by computing the entropy rate based on empirically learned probability distribution [8, 12]. UTrack computes only the first order and second order entropy. It is expensive to calculate even higher order entropy, which may need prior knowledge to determine a probability distribution. Also, we observe that the first and second order entropy can achieve a satisfactory result.

5.2 Non-interactive Process Pruning

Instead of targeting at information-lossless pruning [25, 46], we can afford to remove less interesting data points when coping with our specific goal of user activity tracking. However, it does not mean we do not track other processes. Actually, we keep track of all alive processes that have any interaction with interactive processes or become interactive processes. The processes we pruned are those that do not have any lineage with an interactive process. In general, most processes that are not in a user session are pruned since they are system-triggered events.

For the processes in a user session, if they are not related to any user interactions, they are also pruned. We develop an online algorithm to prune those processes using a bottom up, and backward propagation method. The pruning starts from the leaf process when the process is ended. If the leaf process can be pruned, it is removed from the child list of the parent process, and the parent process will be further checked to see if it can be pruned after the removal of its child process.

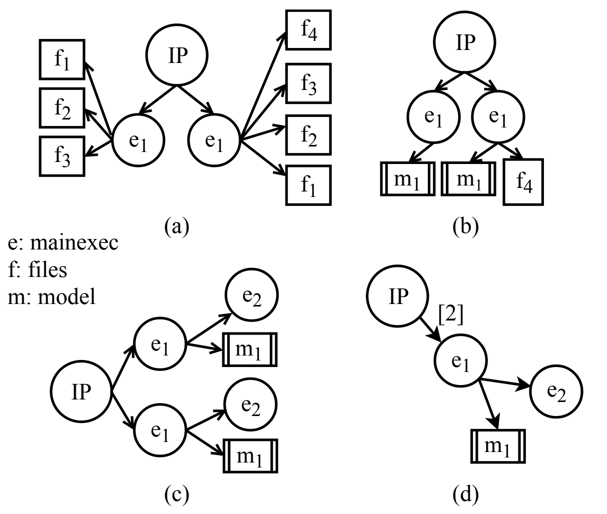

Figure 5: Data Modeling

5.3 Data Modeling

The essence of identifying interactive processes is to find the activities of processes, since they are likely the direct results of the user's operations. Therefore, all activities of the interactive processes are preserved in our user session profile. Due to its long-living and interactive nature, an interactive process usually has many sub-process branches representing user activities. However, these branches could be highly repetitive due to multiple reasons. First, even interactive processes may have periodic routines for updating, synchronization, etc. Second, user activities can be repeated. For instance, a user may run "ls" command many times in a terminal, and many commands intrinsically invokes "ls". Third, there is a large gap between user operations and the interpretations of the computer system. Therefore, a single, seemingly atomic user operation may result in a large amount of low-level events. For example, when opening a Firefox browser, we observe that a significant portion of events are repetitive to serve the same low-level purpose, such as checking the system time or OS version.

There is a large room for the improvement on salience of a user session profile by modeling and compressing the files accessed by processes and the branches of interactive processes, respectively. Based on the observation that many processes with the same executable name and same arguments (e.g., Chrome.exe type=renderer . . . , we call them "mainexec") may access a similar set of files, we are able to model commonly accessed files under a mainexec, and record only the difference. An example is shown in Figures 5(a) and 5(b). We notice that both mainexecs e_1 and e_2 access a common set of files ($\{f_1, f_2, f_3\}$), which can be abstracted by a model (m_1). This model-based technique has also been used in Arnold [11] to reduce instrumentation overhead.

Figures 5(c) and 5(d) show that the session profile can be further compressed if some branches are identical. Interactive processes often have identical branches that could easily overwhelm the auditor. Therefore, we can compress these identical branches by only recording the timing information and the number of occurrences.

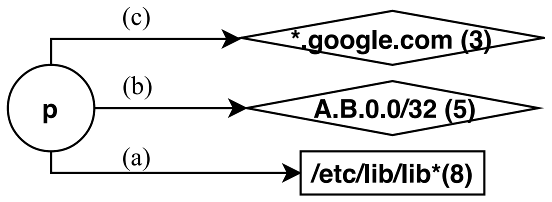

Figure 6: File and IP Abstraction

5.4 Presentation Simplification

Different from conventional backtracking or forward tracking of an attacking incident, the session profile produced by UTrack describes a user session. Thus, the session profile becomes unavoidably larger and cannot be further compressed since all data points carry meaningful information. To better visualize the data for user tracking, we use graphs to present all processes, files, and network connections in a user session. We visualize the session profile generated by UTrack using the dot language, and then apply different level of simplification on the graph.

A fundamental challenge of presenting the session profile on a single graph is that the graph could be very large due to processes accessing a large amount of files or network connections in a long session. To alleviate this issue, we aggregate similar files and network connections when visualizing the session profile graph. For instance, the activities of a process can be represented as in Figure 6. For the files, we find common prefixes of the file names and abstract them with the same prefix. The network connections are either aggregated using the host name of the IP address.The details of the abstraction can be found in Section 6.5. Note that the essential difference between the presentation abstraction and the data reduction/compression techniques is that the actual data model is not changed in the presentation simplification process. Namely, the abstraction does not preserve any resource, but is only used to help the security auditors to have a better view on the data.

6 IMPLEMENTATION AND EVALUATION

6.1 Experiment Environment

We deploy UTrack on 111 hosts of a real enterprise environment, 21 Linux hosts and 90 Windows hosts. An agent is installed in each host to collect and report system events. UTrack itself is written in Java and contains 8.3K LoC. We evaluate the performance of UTrack based on one month of data. Within this period, more than 4 billion events are generated, where 1.65 billion events come from Windows hosts and 2.41 billion events come from Linux hosts. To facilitate the use of history data, we implement a data replayer to replay the data recorded and stored in the database with their original timestamps. With the assistance of the replayer, we are able to replay the one-month data within 30 hours.

6.2 User Tracking

In our one month experiment, we identify 507 user sessions across 111 hosts. Note that the login screen itself is counted as a user session and excluded from our data. Among the total 507 user sessions, only 61 of them are Linux sessions. One reason is that Linux users are less likely to log off or restart their computers than Windows users. Besides, there exist 4 Linux hosts that do not have any user sessions, which means that they are used as servers and no one logs on the hosts through the Linux desktop environment. However, the activities in those servers may be correlated to user sessions in other hosts. On average, each user session lasts 4.6 day. We also observe that Linux sessions are significantly longer (9.1 days) than Windows sessions (3.9 days). More than 100 sessions last beyond the one month period, so they are excluded when we compute the average session lifespans.

Table 1: Servers with the Most Network Connections

Program Name	Number of Instances	User Instance	Mode	Host Type
sshd	134,492	671	Create New	Linux
smbd	8,120	428	Create New	Linux
Postgres	5,152	559	Create New	Windows&Linux
sendmail	1,218	17	Create New	Linux
httpd	874	841	Worker Pool	Linux

For cross-host tracking, we first identify the communication channels. We correlate network events from all hosts by matching 5-tuple attributes, which include local IP, remote IP, local port, remote port, and the network protocol. However, due to port or IP recycling, two network events might be wrongly matched. To avoid such a situation, we add a constraint that two matching events should happen within a small time window. This small window should consider the possible errors caused by asynchronous clocks on different hosts and network resource recycling. In our implementation, we set the time window to 60 seconds, and we recycle the unpaired events after this time window.

In our environment, the number of all ready-to-pair network events stabilizes at around 20,000 to 25,000. We observe that only around 12.3% of network events can be eventually paired, and most of the matched network events (82.4%) are localhost channels. This is reasonable because any communication to the outside world cannot be paired. Even the internal communication may not be identified, since not all computers host an agent in our environment. Another case is the broadcast network events, which have multiple receivers. When the server is working in the worker-pool mode, it may take a non-negligible time to determine the delegated worker, since it needs to go through a network channel matching process. If a worker is found, a virtual process will be created for the requester. However, before the virtual process is created, the network request may have already been partially or entirely handled, because most requests are handled very quickly. Thus, one should record the mapping between the virtual node and the actual node, and migrate the stand-out events to the virtual node once the delegation relation is established.

During the one-month experiment, we observe more than 186 programs that accept network connections, and the top 5 programs are listed in Table 1. The “Number of Instances” column shows the total number of request processing instances we observed. In our environment, since a server frequently runs ”ss” to localhost for system backup, we observe a large number of ssh events. We also find a Postgres database that constantly stores new data from network connections. An Apache server runs the default pre-fork Multi-Processing Module (MPM) to support a worker pool. The “User Instance” column indicates the instances that belong to a user

Table 2: Classification Results

	Interactive	Non-interactive	Total
Classified as Interactive	447 (TP)	181 (FP)	628
Classified as Non-Interactive	5 (FN)	25,746 (TN)	25,751
Total	452	25,927	

session. It shows only a small portion of the cross-host activities can be seen in a user session, since most of the virtual process nodes are pruned due to their irrelevance to user activities.

6.3 User-centric Activity Tracking

To detect interactive processes, we employ an important feature, the "regularity" of activities, which is measured by the first and second order entropy rates on the inter-arrival time of activity batches in a process. In our implementation, we empirically set the threshold of batching (T) as 350 ms. Figure 7 illustrates the CDF of the number of batches a process has when doing the interaction detection, and the number of events a batch has. Both of them are heavily tailed. For clarity, we limit the x axis to be within 100. We observe that 70.7% of processes have only one batch, and 99.1% of processes have fewer than 100 batches. Similarly, more than 78% of batches have fewer than five events, and 97% of batches have less than 100 events. When computing the entropy of the process, we round the time interval to second-granularity to mitigate noise.

Another important feature we use is the lifespan of a process, which describes the time duration from the time the process is created to the time it is ended (or the current time if it is still alive at the time of decision making). The lifespan is a strong indicator of an interactive process. Due to the communicative nature, interactive processes tend to live longer than other processes. Therefore, a large amount of transient processes, especially in Linux hosts, could be filtered out by inspecting their lifespan of milliseconds. Figure 8 illustrates the CDF of process lifespan, which indicates that around 90% of processes have a short life time less than 20 ms.

Our model also considers the context of a process, including the parent process, the number of child processes, and the nature of the parent, as a set of important features. If a process is created by a window manager (e.g., compiz is the default window manager in Ubuntu 16.04), the process is more likely to be an interactive process. We also blacklist 16 types of commonly seen non-interactive processes (e.g., "/etc/update" periodically runs on Ubuntu OSes) to remove unnecessary distractions. It is hard to maintain a whitelist since a process could be sometimes interactive and sometimes non-interactive depending on user operations.

In total, we extract 11 features to build a random forest model based on Weka [15] to predict if a process is interactive. In the training stage, we manually examine and label processes in 50 user sessions (20 Linux sessions and 30 Windows sessions) in a one-day period. An advantage with manual effort is that we can deliberately search the mainexec of a process online and better understand what the process is used for. The machine learning module is triggered when a process's ending event is observed or the process has a sufficient nubmer of activities, including batches, network connections, and child processes.

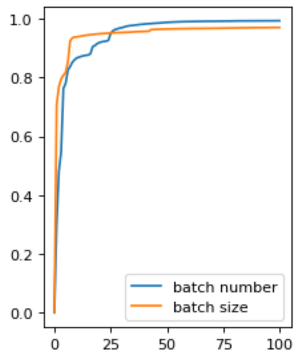

Figure 7: Batches in Processes

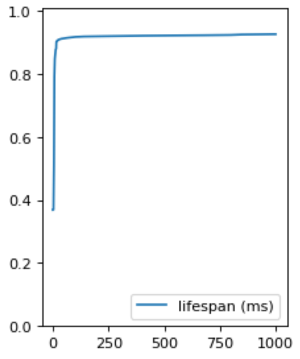

Figure 8: Lifespan of Processes

More than 26,000 processes are labeled after filtering out the processes on the blacklist. We apply 10-fold cross-validation on all the processes, and the evaluation results are shown in Table 2. Our machine learning module has a high accuracy and recall of 99.3%. However, the module has a fairly low precision, which is only 71.1%. Therefore, in a user profile, there are a non-negligible portion of processes that do not really interact with the user. However, even with those false positives, the user profile has been largely reduced since the dominant factors of non-interactive processes are mostly identified and pruned off. We argue that having some wrongly classified processes in the profile is acceptable since the amount of noise created is limited and can be easily identified by the security auditors.

6.4 Data Modeling

We model files accessed by both processes and sub-process branches, and compress them by only recording the deviations from the model. This modeling process is done when the process is ended. In most cases, the interaction detection module also kicks in at this moment. It produces the same results no matter which module runs first, since the modeled processes will be pruned if they or their ancestors are later decided to be non-interactive. On the other hand, pruned processes do not go through the interaction detection stage. In fact, most processes do not live more than 20 ms (as shown in Figure 8), and will be immediately pruned or modeled. In our implementation, we apply data pruning, if applicable, before modeling for higher efficiency. As such, the evaluation results are only applied on the interactive processes and their offspring. In contrast, non-interactive processes are pruned off before any modeling can be done.

We build an FP-Tree [16] to model the commonly accessed files of the same mainexec on each host. The FP-Tree is frequently used to mine association rules from a growing data. We set the Minimum Support Threshold (MST) to 0.3, so that files with frequency less than 0.3 are discarded from the tree. The FP-Tree no longer changes after the training period. At this stage, new processes with the same mainexec can be modeled using the Tree. Since many processes may have the same process branches that exhibit exactly the same system behaviors, we compress the same branches into one and record the number of occurrences. Some processes may have a random token in the mainexec, such as the Chrome renderer processes. We handle them in a case-by-case manner. Similar to data pruning, our online branch modeling algorithm adopts a bottom up approach, which starts modeling from the leaf processes and propagates back to the parent process if no leaf process is alive. When a parent process notices that multiple child processes have the same model, it merges these child processes and records the occurrence of the model. The model of each process is represented in an XML-styled structure, which stores the information of files, remote IPs, and mainexec of itself and its offspring. Note that the backward propagation in our user session model stops at the interactive processes. This is because the model becomes increasingly large in lower-depth process nodes due to the large number of child processes. Also, it does not provide any help on compressing the data, because the process models at these levels are rarely identical and thus hardly compressible.

In the 507 user sessions, 8,382 interactive processes and 176,822 other processes (i.e. the child processes of the interactive processes) are identified. After modeling the branches, more than 71% of the processes are compressed, leaving us 8,382 interactive processes and 50,394 of their child processes. All 58,776 processes access more than 1.2 million files. After data modeling, the number of files reduces to 502,446 (around 60% reduction), where a file model is counted as one file.

On average, each user session has about 116 processes, which is a small number considering that a user session can last for several days. However, the number of accessed files is large, almost 1,000 files per session profile. We observe that the majority of files come from process initialization, since a new process can easily read hundreds of files during initialization. Although this initialization process can usually be modeled, our experiment period may not cover enough instances of the processes, so all the files are preserved. When UTrack runs for a sufficiently long time, these processes can also be modeled. Our implementation on presentation simplification, which is designed for purely presentation with some information loss, can partially remedy this problem.

6.5 Presentation Simplification

We can further simplify the graph describing the user session profile to provide a better view for the auditors, especially when a process is opened and many configuration files are read at once. Furthermore, there are cases that a large amount of temporary files with random names are created, so these files cannot be modeled at all. To tackle these problems, we manage to abstract the files and network channels into few nodes while preserving sufficient information for understanding the whole process. Note that the complete data is not lost in this abstraction step, so if needed, an auditor can look into each abstraction for complete information.

Similar endeavors have been made in [17] to model the behavior of containers that run the same services. However, in their implementation, except for some special types of files, all other files are collapsed into one abstract, which is too aggressive and tends to lose important information. For example, when we have 10 files prefixed with "/A/B/C/" and 1 file named "/A/D/EFG", a preferable way might be to preserve both "/A/B/C/*" and "/A/D/EFG" instead of collapsing all files into "/A/*".

6.5.1 File Abstraction. When there are a large amount of file access that cannot be modeled, we may need to sacrifice data accuracy in file abstraction in exchange of a more succinct presentation. Toward this end, we build a *trie* (a.k.a. prefix tree) for all the files that are accessed by a process. In a trie, each node stores a single character, and two file names with a common prefix share a single path until the end of the common prefix is reached. By carefully trimming the tail branches of the tree, we can achieve a higher degree of condensation and keep the output graph concise. However, due to trade-off between the amount of preserved information and the degree of abstraction, we can only benefit from the abstraction when the gain is deemed larger than the cost. In our case, the gaining is defined by the number of files that can be abstracted when folding all nodes under a single node in the trie, and the cost is the loss on the filename characters. Specifically, the gain is defined as $\sum_{i=1}^{n} g_i$, where n is the total number of files that can be abstracted, and g_i is the preserved information for each of the file, which is set as $\frac{l}{L} + f$, where l denotes the number of complete directory level of the abstract file name, L denotes that of the complete file name, and f represents a fraction of the preserved part of the last level of the file name. For example, when abstracting the file "/A/B/C/DEF" to "A/B/C/D*", it is $\frac{3}{4} + \frac{\frac{1}{3}}{4} = 0.83$. Note that we stress the directory depth instead of the length to mitigate the effect of long file names. We develop a greedy algorithm to trim the trie in an iterative manner. In each iteration, we abstract the files with the maximum gaining until the number of files is within a *max_file* threshold, which specifies the number of files that we prefer each process not to exceed in the presentation. Another scenario to stop the iteration is when the gain of abstracting a set of files is less than a threshold (τ). Note that τ cannot be less than 1, otherwise a single path will be abstracted with no gaining. Both max_file and τ can be adjusted to balance the precision and size of the graph.

6.5.2 Network Abstraction. Network events are also hard to model since user-driven network channels cannot be easily predicted, and thus cannot be modeled accurately. Therefore, it is important that we abstract the network behaviors. We choose to model the remote IP addresses. First, we look up the domain name of a remote IP address. If the domain name can be found, we abstract IP address with the same top-level domain and secondary-level domain, since these two levels of domains are usually sufficient to identify a remote host. Now an abstraction may look like "*.google.com" or "*.Facebook.com." In case the remote hostnames cannot be found by a reverse DNS lookup, we adopt a network mask approach similar to [17]. Specifically, we abstract the class B and class C subnets to abstract most IP addresses in a similar manner to file abstraction.

6.6 Graph Presentation

Figure 9 depicts a simplified user session profile we identified from our network environment. In Figure 9, the two different colors indicate two hosts. Processes, files, and remote IPs are represented by ovals, squares, and diamonds, respectively. The complete graph has a total of 323 nodes, including processes, abstracted files and remote IP addresses. For simplicity, we omit most unimportant files from the graph with one exception of the "outlook.exe" process. We illustrate the 5 abstracted files read by "outlook.exe" in the graph to give a basic idea of how the files look like after the abstraction step. For other processes, we put the number of abstracted files

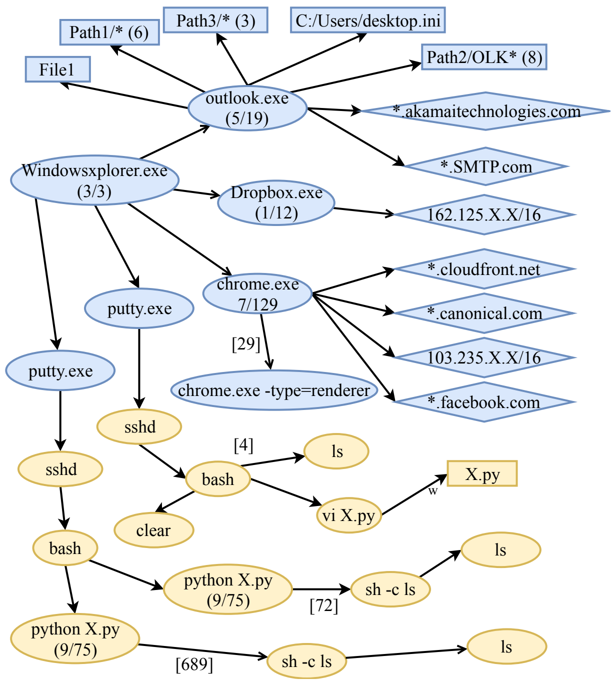

Figure 9: Example User Profile

Path1:C:/Users/X/appdata/local/microsoft/windows/temporaryInternetfiles/content.IE5
Path2: C:/Users/X/appdata/local/microsoft/outlook
Path3: C:/Users/X/appdata/local/TEMP

File1: C;/program files/common files/system/ado/msadox.dll

and the number of total files inside the process node. The case of a process without a number indicates that the files are completely modeled. Meaningful files are preserved as nodes on the graph. The timing information is not included in this figure due to the space limit. Besides, timing is not critical in understanding the figure. The number of abstracted branches is shown in brackets on edges.

From the figure, we can easily find the user's activities inside the enterprise network. The user logs in the system on a Windows host and the session lasts for six hours. The session spans two hosts through interactive ssh connections using putty. On the Windows host, the user browses the Internet via Chrome and uses Outlook for emailing. The user then logs on a remote Linux host to edit and run a python program "X.py" (the file name is anonymized), which further runs the "ls" program many times. In general, a graph-based session profile presentation can be easily understood by a human auditor, and provides important insights on the activities of a user.

7 USE CASES

Many more UBA features can be directly applied to UTrack for anomaly detection. For example, one can audit the roles (user accounts) that a user has been playing in the network from the user profiles and identify higher-level inconsistencies. For instance, one cannot be both "Alice" and "Bob" in the same session profile. Besides providing a foundation of UBA systems, there are many other use cases that can be built on top of UTrack. For example, it can be used in forensics analysis to study the behavior of attackers (such that the attacker becomes the POI) and reveal more seemingly benign behaviors which are in fact part of the cyber kill chain. UTrack can also be used to determine the value of files (by the amount of time an employee spent on a file) for backing up digital asset. This is particularly useful in fighting ransomware.

8 RELATED WORKS

User Tracking and UBA User or user activity tracking has been extensively studied in different contexts and various techniques have been proposed. One typical scenario is web user tracking through different measures [1, 3, 28]. User behavior tracking for the security purpose drives UBA, where user accounts are no longer the single indicator of who an incident is performed by. Nowadays, many security companies have announced UBA tool integration or plan to develop UBA in their systems [4, 19, 20, 35, 42].

UBA consists of two steps. The first is to model normal user behaviors, and the second is to detect abnormal users by examining how deviated they are from normal users. There can be many metrics, algorithms, or machine learning models being used to identify an abnormal user [33, 35]. Contemporary UBA mostly models users based on basic patterns or statistics, for example, several basic statistics, such as total upload bytes and total download bytes of a user [33]. However, to detect more sophisticated attacks, it is vital to ensure high accuracy and descriptiveness of user activities.

Log Audit Log audit has been used in many fields of security research, such as forensics analysis [22, 23, 27], intrusion recovery [14, 21], and intrusion detection [10]. One of the most widely adopted log levels is the OS level, where the basic units are process, files, sockets, etc. The reason is that the OS level maintains high fidelity of states of the entire system, as well as incurring acceptable CPU and storage overhead [22]. There are previous works focusing on the reduction of the storage overhead while not losing much information [25, 46]. Besides, there are also previous works that attempt to increase data granularity based on OS level logs [24, 26]. One important use of log audit is to understand an attack, especially more sophisticated attacks (APT attacks) or unknown attacks. Security experts rely on the logs to determine how an attack happens [22, 24, 27], as well as its impact on the system [23]. They capture the causal relationship among processes, files, or sockets, and reconstruct the provenance of an attack and its ramification. HERCULE [30] leverages community discovery algorithms to identify attacks based on the fact that the attack activities belong to the same community in a graph. [6] logs events at the proxy and focuses on parsing traffic from application protocols like SQL.

User Interaction Detection The detection of bot generated data (system-triggered) from human-generated data (user-triggered) is a long-studied subject that has applications in many fields. Generally there are two types of detection. One is the active detection, such as CAPTCHA [44], which is easy to implement, (arguably) more accurate, but intrusive. The other type is the passive detection, which relies on processing log events to detect abnormal behaviors. The related previous works include detecting game cheaters through Human Observational Proof [13], bots in online social networks [7–9, 43], detecting malicious web bots/crawlers, Google reCaptcha [34], and malicious crawler detection [45]. There are some significant differences between these techniques and ours. A major one is that they have specially tailored data input. For example, user agent, cookie lifetime in Google's reCaptcha [34], a user account favored access log system in [13, 45], or side information such as social graph [7, 9, 13, 43].

9 CONCLUSION

This paper presents UTrack, a novel user tracking system that connects events under different user accounts and from different hosts to form a novel holistic user session profile. UTrack enables a system auditor to easily find out the activities of users inside enterprise networks. UTrack associates the activities of a user effectively by identifying a session root and then following both the local process lineage and the network control flow of the session root. To achieve scalability and salient description, UTrack employs an interaction detection module to sift out the most relevant events that result from users' interactions, and models common file and activity patterns. Our evaluation in a real enterprise environment of 111 hosts shows UTrack's effectiveness on producing accurate and concise user session profiles for system auditors to use.

REFERENCES

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM CCS*.

[2] Animashree Anandkumar, Chatschik Bisdikian, and Dakshi Agrawal. 2008. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 36. 133–144.

[3] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. 2006. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *WWW*.

[4] BALABIT. 2015. Privileged Account Analytics - User Behavior Analytics Security Solution. https://www.balabit.com/privileged-account-analytics.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling.. In *USENIX OSDI*.

[6] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2017. Transparent Web Service Auditing via Network Provenance Functions. In *Proceedings of the 26th International Conference on World Wide Web*. 887–895.

[7] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. 2012. Aiding the detection of fake accounts in large scale social online services. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. 15–15.

[8] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. 2010. Who is tweeting on Twitter: human, bot, or cyborg?. In *Proceedings of the 26th ACM Annual Computer Security Applications Conference*. 21–30.

[9] George Danezis and Prateek Mittal. 2009. SybilInfer: Detecting Sybil Nodes using Social Networks.. In *NDSS*. San Diego, CA.

[10] Dorothy E Denning. 1987. An intrusion-detection model. *IEEE Transactions on software engineering* 2 (1987), 222–232.

[11] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic Systems.. In *USENIX OSDI*. 525–540.

[12] Steven Gianvecchio and Haining Wang. 2007. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 307–316.

[13] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. 2009. Battle of botcraft: fighting bots in online games with human observational proofs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 256–268.

[14] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. 2005. The taser intrusion recovery system. In *ACM SOSP*. 163–176.

[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* (2009).

[16] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, Vol. 29. 1–12.

[17] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *NDSS*.

[18] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. 2017. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *26th USENIX Security Symposium*. 487–504.

[19] IBM. 2016. IBM QRadar User Behavior Analytics. https://www.ibm.com/cz-en/marketplace/qradar-user-behavior-analytics.

[20] Johna Till Johnsons. 2015. User behavioral analytics tools can thwart security attacks. http://searchsecurity.techtarget.com/feature/User-behavioral-analytics-tools-can-thwart-security-attacks.

[21] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution.. In *USENIX OSDI*. 89–104.

[22] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. *ACM SOSP* (2003), 223–236.

[23] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality.. In *NDSS*.

[24] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*.

[25] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. 1005–1016.

[26] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACM ACSAC*. 401–410.

[27] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of NDSS*, Vol. 16.

[28] Jonathan R Mayer and John C Mitchell. 2012. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy 2012*. 413–427.

[29] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. 2019. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy*. 1137–1152.

[30] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 583–595.

[31] Patrick Reynolds, Janet L Wiener, Jeffrey C Mogul, Marcos K Aguilera, and Amin Vahdat. 2006. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web*. 347–356.

[32] Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia. 2012. Precise, scalable, and online request tracing for multitier services of black boxes. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2012), 1159–1167.

[33] Madhu Shashanka, Min-Yi Shen, and Jisheng Wang. 2016. User and entity behavior analytics for enterprise security. In *2016 IEEE Big Data*. 1867–1874.

[34] Suphannee Sivakorn, Jason Polakis, and Angelos D Keromytis. 2016. I'm not a human: Breaking the Google reCAPTCHA. *Black Hat,(i)* (2016), 1–12.

[35] Splunk. 2015. Splunk User Behavior Analytics. https://www.splunk.com/en_us/products/premium-solutions/user-behavior-analytics.html.

[36] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N Chang. 2009. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities.. In *USENIX ATC*.

[37] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. 2006. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 34. 3–14.

[38] Mike Tierney. 2015. The Rise of User Behavior Analytics. http://www.veriato.com/company/blog/veriato-blog/2015/12/15/the-rise-of-user-behavior-analytics.

[39] Roy Hodgman Tod Beardsley. 2015. RAPID 7 Research Report: Understanding User Behavior Analytics.

[40] Trustwave. 2015. Trustwave global security report. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[41] Melissa Turcotte and Juston Shane Moore. 2017. Technical Report LA-UR-17-21663: User Behavior Analytics.

[42] VARONIS. 2016. User Behavior Analytics. https://www.varonis.com/user-behavior-analytics/.

[43] Bimal Viswanath, Ansley Post, Krishna P Gummadi, and Alan Mislove. 2010. An analysis of social network-based sybil defenses. *ACM SIGCOMM Computer Communication Review* 40, 4 (2010), 363–374.

[44] Luis Von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. 2008. recaptcha: Human-based character recognition via web security measures. *Science* 321, 5895 (2008), 1465–1468.

[45] Shengye Wan, Yue Li, and Kun Sun. 2017. Protecting Web Contents against Persistent Distributed Crawlers. In *IEEE ICC*.

[46] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*.