SemFlow: Accurate Semantic Identification from Low-Level System Data

Mohammad Kavousi [¶], Runqing Yang [†], Shiqing Ma [‡], Yan Chen [£]

[¶,£] *Northwestern University*, [†] *Zhejiang University*, [‡] *Rutgers University*,
[¶]`kavousi@u.northwestern.edu`, [†]`rainkin1993@zju.edu.cn`,
[‡]`shiqing.ma@rutgers.edu`, [£]`ychen@northwestern.edu`

Abstract. Forensic analysis, nowadays, is a crucial part of attack investigation in end-user and enterprise systems. Log collection and analysis enable investigators to rebuild the attack chain, find the attack source and possibly rollback the damage made to the system.

However, building the full attack chain is often time-consuming and error-prone. The reason is that existing audit systems cannot provide high-level semantics for low-level system events. To address this issue, we propose SEMFLOW, to accurately identify semantics for system events. Specifically, we generate signatures to link low-level system events to a particular high-level application behavior during an offline training phase. Then, during the labeling phase, our realtime data collector matches the generated signatures against audit logs and labels individual system-level events with high-level semantics.

Our evaluations show that in at set of 16 selected popular applications, our system can effectively identify semantics of certain system-level data while maintaining less than 4 percent of overhead on the CPU and memory.

Keywords: security· system security · semantic detection · provenance graph · living-off-the-land

1 Introduction

Large enterprises are increasingly being targeted by Advanced Persistent Threats (APTs). One of the main goals of APTs is obtaining and exfiltrating highly confidential information, e.g., APT1 [4] stole hundreds of terabytes of sensitive data (including business plans, technology blueprints, and test results) from at least 141 organizations across a diverse set of industries. By avoiding actions that would immediately arouse suspicion, attackers can achieve dwell times that range from weeks to months. Alternatively, Living-off-the-Land (LotL) tactics take advantage of native tools existent on a system in order to perform lateral movement and gain persistence of APT threats in operating systems. During Q3 2018 to Q3 2019, Symantec saw a 184 percent increase in blocked Windows PowerShell scripts. 87 percent of such attacks executed the PowerShell script through cmd.exe or Windows Management Infrastructure (WMI)[5].

To combat such threats, the notion of data provenance has been applied to traditional system event audit systems and have proven invaluable in detection and investigation. Highly confidential information is usually represented as

file objects and any operations related to such file objects are recorded in system event audit logs. Data provenance analysis systems parse individual system events into provenance graphs that encode the history of a system's execution sequence. Such provenance graphs allow investigators to trace the data flow of highly confidential file objects. By leveraging such capability, security systems can identify attack steps and the involved objects, when the content of confidential files is read and is finally sent out to the opposition. Essentially, provenance graphs can help identify the whole attack chain of an adversary; beginning from the initial compromise up until their lateral movement and business damage. In this paper, we focus on attack types which use benign software to perform some of, or the whole attack.

Unfortunately, existing system event audit systems cannot provide application-level semantics for file events, which leads to inaccurate results on almost all existing data provenance based systems. For instance, WinSCP, a popular file transfer software on Windows, might read the confidential private key file for two different purposes: authentication, and upload. However, existing audit systems provide two totally equivalent events (i.e., two ReadFile events referring to the same file object) for such two scenarios. Without application-level semantics of ReadFile events in this example, a normal user logging onto the SSH server using WinSCP by reading the key file for authentication would be mistakenly considered as a data exfiltration attack. This would lead to the problem that provenance-based systems would lack such kind of information due to the semantic gap between low-level system events and high-level application operations.

Some prior works have been able to provide high-level application semantics for system events [13, 28, 22]. Although they demonstrated great potential, they suffer from several limitations:

1) **Requiring invasive instrumentation**. Some use instrumentation techniques [23] to intercept application-level APIs and correlate such semantics to system events. Such systems introduce notable performance degradation and instability, which is not applicable in enterprise environments. Furthermore, they highly depend on application-specific libraries (e.g., instrumentation on browsers based on Chromium cannot easily be extended to other applications).

2) **Inaccuracy**. Some works rely on heuristics (e.g., timestamp-based correlation [39]) to correlate application-level logs to system events without instrumentation. However, such heuristics cannot work in complex scenarios (e.g., logs generated by multiple individial behaviors occur in parallel in the background), which leads to inaccurate results.

3) **Post-processing**. Existing data provenance based works [29, 15] consume system events and detect attacks in realtime, which requires the collector to provide the semantics of system events in realtime to enhance those works, resulting in the need for labeling before the data is consumed. However, existing works [22] rely on complex behavior models (e.g., behavior graph model whose vertexes are system call and edges are causal relationships between system calls) to provide high-level semantics accurately, which makes them non-realtime.

Problem statement. *The main problem tackled in this paper is providing application-level semantics for file and process/thread related events under user interactive scenarios without instrumentation accurately in realtime.*

In this work, we argue that existing system event-based works can be dramatically improved through identifying high-level application semantics of individual system events. To achieve that goal, we present SEMFLOW, an instrumentation-free framework that takes advantage of the native Windows audit system for identifying high-level application semantics of systems in realtime, with no need for post-processing of system data. Our methodology is inspired by analyzing the program callstack which reveals its API call chain, and observing that parts of the chain are designated to perform a specific task. The main secret that we can provide such semantics without complex binary instrumentation, is the return address of each frame in the application callstack, which we can refer to as the line of code. In other words, the execution sequence that caused the system-level event to be emitted. We argue that in most cases, the sequence of function calls leading to emitting a specific event are unique to each behavior and can accurately identify it.

For our approach, we will perform training on different sets of behaviors in various applications, generate signatures and after that, define a set of user interactions that can produce the same behavior; so that it can be done automatically for the later versions of the same program.

The training does not need to be done on the user end. Hence, it can be done on high-end servers as demanded, reducing the time of signature generation for each behavior on a specific program binary.

To implement and evaluate our approach, we have collected about 150 datasets from different application executables performing different behaviors. Part of which will be used to train and the rest for evaluation purposes.

Our evaluations show that we result in a very small percentage of false positives labeling individual events. In addition, since we use set matching and all of program behavior signatures can be loaded into memory, the overhead of performing such labeling is negligible, compared to collecting unlabeled events. We also show that our method can be applied to systems already using an existing intrusion detection system (IDS); as our system labels individual data/control flow events and performs such labeling in realtime.

In summary, this paper makes the following contributions:

- We recognize the gaps of existing event identification systems and try to fill by identifying high-level application semantics of events.

- We propose a novel behavior model based on callstacks to identify high-level semantics of individual events accurately and efficiently.

- We propose an instrumentation-free audit system, SEMFLOW, which provides the semantics of popular security-related file and control-flow events in realtime, and we evaluate SEMFLOW by providing cases of potential real-world attacks.

2   Background and motivation

In this section, we discuss the concept of forensic analysis and building the attack chain, and the significance of labeling events as certain activities.

2.1   Motivating example

A startup company purchased a virtual public server (VPS) to host its public website using Apache web server. Because of budget limitations, the purchased service only allows one communication token to the server. Thus, the whole web team shared the same public and private key files generated by the RSA algorithm (`pub.rsa, pri.rsa`) to communicate (via SSH) with the remote server whose IP address is `x.x.x.x`. Due to the importance and secrecy of these keys, the company forbids its usage in other scenarios. A daily routine for the team was to log onto the server using the communication keys via WinSCP program and update the files hosted under the root HTTP folder, `www`.

   The company receives an alert from the VPS company about a possible data leakage. Specifically, a few files were hosted publicly under the folder `www`, and they verified that one of them was the private key `pri.rsa` used to communicate with the VPS machine. By checking the server communication log, they also confirmed that the only SSH connections were from the company machines. To investigate how the key got leaked, the startup company started to perform forensics on machines used by the web team.

2.2   Existing Audit Systems and Forensic Systems

Figure 1 (I) shows a typical provenance graph generated for different team members belonging to the web team in the motivating story. Note that in this figure (and the rest of the paper), we use *diamonds* to denote *sockets*, *boxes* to denote *processes*, and *ovals* to denote *files*. As we can see, all members seem to be suspicious, because all of them have an information flow from the private key file to the remote server. A deeper investigation shows that the reason is because the private key will be used for authentication purposes when using WinSCP to update files on the server.

   This raises the problem that two totally different behaviors (i.e., authentication and uploading private key files to the server) can generate identical provenance graphs, which interferes with the investigation process and prevents us from effectively and efficiently identifying the attacker machine. The root reason is we do not know what the files are used for when the same operations happen (i.e., read in this case, for authentication, and uploading). We refer to this as the **semantic gap** problem, meaning that low-level provenance data lack semantic information (i.e., what the files are used for) from applications. This is a common issue for existing provenance graph-based forensic systems, leading to inaccurate results [20, 26, 12].

2.3   Our System

In this paper, we propose a provenance graph edge tagging technique SEMFLOW, which solves the semantic gap problem by leveraging the (user space) callstack information provided by Event Tracing for Windows (ETW) system to tag the
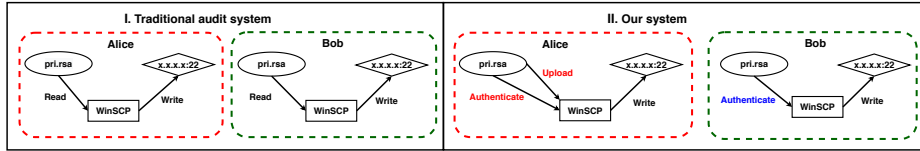
Fig. 1. Attack scenario in 2.1 without (left) and with (right) semantic identification
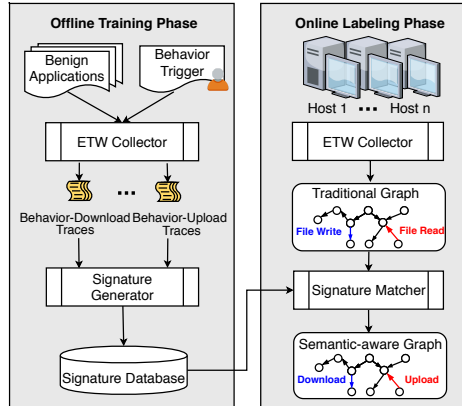

Fig. 2. Overview of SEMFLOW

provenance graphs generated by using low-level information. SEMFLOW first analyzes the callstack information of each ETW event and generates a "signature" for these events based on the callstack, which represents the high-level semantics (e.g., uploading or authentication). When a system event is produced, SEMFLOW will tag individual edges in the provenance graph based on generated signatures, so that events with the same event type can be distinguished.

Figure 1 (II) shows the graph generated by SEMFLOW for the aforementioned example. In the figure, we use blue and red colored texts to represent high-level semantic behaviors labeled by SEMFLOW for the benign scenario and the malicious scenario, respectively. Based on the tag, administrators can easily see the private key is uploaded using Alice's computer.

2.4   Threat Model

In this paper, we consider the OS kernel and auditing system (i.e., ETW) as part of the trusted computing base (TCB). We assume that the OS kernel is well protected by existing techniques [3, 1]. Thus, the audit logs cannot be modified by attackers. Our threat model is as reasonable and practical as the models of previous forensic works [9, 8, 12]. This paper targets attacks involving benign applications, including insider attacks leveraging benign applications' functionalities to perform malicious behaviors and attacks injecting malware onto the machine to hide malicious behaviors under normal behaviors to bypass the Intrusion Detection Systems (IDS).

	ProcessId	ThreadId	EventName
(1)	7068	716	ThreadCreate

	Frame	Binary	Location
	...		
	3	ntdll.dll	NtCreateThread + 0x54
	...		
	16	xul.dll	BackgroundFileSaver::Init + 0xe8
(2)	...		
	30	xul.dll	nsHttpChannel::ProcessResponse + 0x34
	31	xul.dll	nsHttpChannel::OnStartRequest + 0x2c
	...		
	38	xul.dll	MessageLoop::RunHandler + 0x28

Fig. 3. An example of a low-level event log with its callstack

3   System Design and Implementation

Figure 2 illustrates our system architecture—the workflow of our system proceeds in two phases: offline training and online labeling. The goal of the offline training phase is to generate behavior signatures from low-level events. Specifically, we analyze why and how we leverage callstacks to represent the high-level semantics of low-level events (Section 3.1), and use the ETW collector (Section 3.2) to collect low-level system events and use *Signature Generator* to analyze the collected information to generate behavior signatures (Section 3.3). In the online labeling phase, our system utilizes the collector on each Windows host for audit logging. When a low-level event is generated, *Signature Matcher* will decide to label events based on generated behavior signatures (Section 3.4), which can help investigators distinguish between benign behaviors and malicious ones.

3.1   Assumptions and Observations

In this subsection, we first provide an example of a system log along with its callstack, and then discuss why callstack is important for semantics identification.

SEMFLOW configures a native low-level data collection tool on Windows to collect the callstack for each low-level event, which is essential to represent high-level semantics for low-level events.

An example of logs along with event callstack Figure 3 shows an example of a low-level event collected by SEMFLOW. The first block (1) includes the basic information of the event, including the process identifier (ProcessId), thread identifier (ThreadId), and the event name (EventName) (other fields are omitted). The second block (2) shows the above event's callstack, which is a function callstack with top ones (e.g., *Frame* 31) being callers and lower ones (e.g., *Frame* 30) being callees. Each entry has three fields, a frame number, a *Binary* field representing which binary file (e.g., Dynamic Linking Library on Windows) the function is in, and the *Location* fields denoting the resolved symbolic names and offset values. The data format of the *Location* field is *Class*::*Function* + *Offset*. The *Class* shows the C++ class in which the *Function* is defined, and the *Offset* represents the offset relative to the *Function* in which the lower function

Table 1. Part of Firefox File Download Signature

	Event Class	File Paths (only used for verification)	Signature type	Callstack
1	Create Thread	-	init	
2	Write File	`C:\10MB.zip.part`; $Temp$`\VQkG8r+z.zip.part`	during	Hash
3	Rename File	$Temp$`\VQkG8r+z.zip.part`	init	values
4	Write File	`C:\10MB.zip:Zone.Identifier`	during	of
5	Write File	$AppData$`\`...`\cache2\`...`\cachefilename`	during	callstacks
6	Write File	$AppData$`\`...`\places.sqlite-wal`	finalize, contextual	

is invoked. Taking *Frame* 31 as an example, `OnStartRequest` is defined at the class `nsHttpChannel`, and the lower function `ProcessResponse` in *Frame* 30 is invoked at the offset 0x2c relative to `OnStartRequest`.

The Essence of CallStack Existing provenance-based systems only leverage event information (i.e., block 1 in Figure 3). However, different behaviors might generate the same event information; causing the semantic gap issue. The fundamental reason is that system events are too low-level and general to capture the semantics of high-level application behaviors. Thus, looking for logs that contain more semantics than system events is necessary. We observe that *the combination of callstack with their system event effectively reflects the implementation logic of a high-level application behavior*, which is useful to solve this problem. Take the event in Figure 3 as an example. The event is generated by Firefox initiating a file download from the Internet. In frame 16, the `BackgroundFileSaver::Init` function shows that this event is created to save a file in the background. Frames 30 and 31 show that the file saving operation is invoked via an HTTP channel. Intuitively, we come up with the idea to leverage callstack to recover high-level semantics of low-level events. Table 1 shows part of the Firefox file download signature. First row creating the thread to download the file, second one assigning a temporary file name, third one renaming the temporary file name after the user assigns a file name, fourth and fifth filesystem-related and caching operations, lastly, adding the downloaded file to the downloads library.

Based on our analysis of the programs and empirical results, we get the conclusion that a single event along with its callstack can be used to recover semantics for most behaviors. Based on such conclusion, we propose our semantic labeling system for low-level events.

3.2 ETW Collector

We build SemFlow upon ETW, a lightweight auditing system on the Windows platform, which has been widely adopted by many academic papers and industry products. In addition to the events pointed out in Section 2, SemFlow configures ETW to collect the callstack for each low-level event. Then, we iterate through the callstack and record the addresses we observe for each frame. Hence, at the end, we would be left with the basic event information, along with its execution callstack (as shown in Figure 3).

3.3 Signature Generator

The signature generator module takes traces collected from different behaviors of different applications when triggered, and generates signatures for selected

behaviors. We use two terms to describe different types of traces extracted from applications:

- *Matching Traces:* Execution traces that contain all or partial events (represented by logged events) related to the behavior.
- *Non-Matching Traces:* Execution traces that do not contain the operations related to the behavior.

The signature generator acts as a training module which goal is to identify unique events related to an operation, by removing out the events that are also happening when the target operation is not taking place. It then generates a unique hash for each unique event it finds in the matching traces. The final signature related to a behavior, will be a set of hashes representing each event's application-level callstack; which will be passed to the *Signature Matcher* for labeling.

Algorithm 1 Signature Generation Algorithm

Input: Matching and Non-Matching Trace Sets for an Operation Within a Process, and Path to the Process– $S_{matching}$, $S_{non-matching}$, $path$
Output: Signature added to $ProcessSet$
Initialize: $B_{Matching} \leftarrow \emptyset$, $B_{NonMatching} \leftarrow \emptyset$
Entrypoint: The Get-Signature Function

1: **function** FIND-NON-MATCHING(S_{input})
2:     **for** each Trace $T_i \in S_{input}$ **do**
3:         **for** each Event $U_i \in T_i$ **do**
4:             $B_{NonMatching}.add(U_i)$
5: **function** FIND-MATCHING(S_{input})
6:     **for** each Trace $T_i \in S_{input}$ **do**
7:         **for** each Event $U_i \in T_i$ **do**
8:             $B_{Matching}.add(U_i)$
9: **function** GET-SIGNATURE
10:     Find-Non-Matching($S_{non-matching}$)
11:     Find-Matching($S_{matching}$)
12:     $Signature \leftarrow B_{Matching}.removeAll(B_{NonMatching})$
13:     $ProcessSet.put(SHA256(path), Signature)$

Algorithm 2 Matching Algorithm for Signatures

Input: Event Stream– e and Process Creation Events– p
Output: Event Set Labeling $B_{EventList}$
Initialize: $SigList \leftarrow \emptyset$, $ProcessSet \leftarrow$ all processes with signatures

1: **function** ON-PROCESS(p)
2:     /* Find the executable location */
3:     $path \leftarrow find\_path(p)$
4:     /* Calculate the hash of executable */
5:     $hash \leftarrow SHA256(path)$
6:     **if** $ProcessSet.contains(hash)$ **then**
7:         $Siglist.add(ProcessSet.get(hash))$
8: **function** ON-EVENT(e)
9:     **if** $SigList.contains(e.callstack)$ **then**
10:         $Label(e)$

3.4   Signature Matcher

After building signatures for different behaviors in different applications, the *Signature Matcher* matches the signatures against each callstacks in the log stream. As for the process for matching signatures with the incoming events

from ETW, upon creating a process, we see whether the signatures for the same binary exist in our signature database. If they do, then for upcoming events for the process, we search in the process-specific signature to see if we can find the same callstack hash in the event set in the signature. If we do find the event with similar callstack, we then label the event according to the signature. Each event in the signature is associated to a behavior which we have extracted from training. As the events enter the signature matcher, the set matching helps us to label (or not to label) each individual event.

Table 2. List of intercepted operations in SEMFLOW

Application	Operations
VLC Media Player	Video Playing
FileZilla	File Upload/Download
WinSCP	File Upload/Download/Copy
Notepad, Sublime Text	File Save
Skype	File Upload/Download
WinRAR	File Compression
Firefox, Chrome, Edge	File Upload/Download
Cobiansoft Cobian Backup	Routine Backup
Outlook	File Upload
Windows Explorer	File Copy/Move
Microsoft Word	Type and Save
TeamViewer	File Upload/Download
Adobe Reader DC	PDF Open and Fill

4   Implementation

In this section, we discuss how we implement each module discussed in Section 3.

4.1   Signature Generator

Algorithm 1 illustrates how signatures are generated for a certain behavior. Taking the matching and non-matching traces, we start running Algorithm 1 by triggering the `Get-Signature` function as entrypoint (line 9) to extract signatures for an activity. It starts by finding all events related to non-matching behaviors inside the `Find-Non-Matching` function (line 10), and all events related to the behavior in the `Find-Matching` function (line 11). The algorithm finds events that occur in matching traces but not the accumulated non-matching events; and finally extracts the unique set of events related to the operation extracted from matching traces, excluding events from the traces when the operation was not happening.

For example, for Firefox file download signature, we run the algorithm from a simulation of different file downloads as matching behaviors. Browsing, changing settings, and playing other functionalities are considered as non-matching behaviors. The algorithm starts by accumulating all non-matching events (line 4), collectively putting them in $B_{NonMatching}$. From three traces collected from matching behavior as $S_{matching}$ and five traces collected from non-matching scenarios as $S_{non-matching}$, line 8 then adds matching events using the same procedure. In the end, in line 12, we remove all the events that also happen
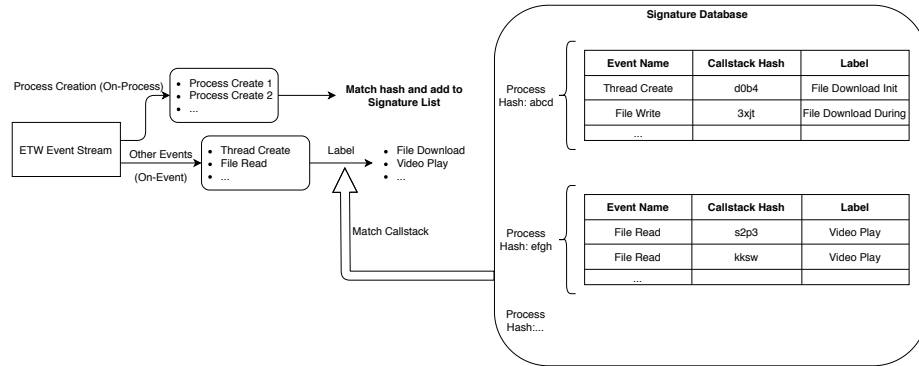
Fig. 4. Signature Matching Mechanism According to Algorithm 2

in non-matching scenarios, allowing us to be left with unique events related to the certain behavior. We use the hash of the process to be able to later identify the binary (described in the next section). We will also propose a method for verifying signatures in Section 5.

4.2   Signature Matcher

Algorithm 2 shows how the matching takes place. Upon starting a process, the program finds the executable path (line 3) and generates a hash from the binary (line 5). Then, we see if we have any training data based on the process (line 6), then we load the signatures for that process into memory.

Upon receiving an event (line 8), the program determines if it maps to any behavior (line 9) that we have signatures on. Then finally, the labeling takes place, and the event can move forward in the stream (line 10). Figure 4 shows an example of signature database and its usage in the ETW event stream, as well as the mapping between Algorithm 2 and the diagram. Hashes from process executables are generated after receiving each *Process Start* event. If it matches any of the recorded hashes in the signatures, then the PID is recorded and the corresponding signature is loaded into the memory. Then if events emitted by any of the recorded processes would match any of the events in signatures, then the event is labeled.

5   Experiments and Evaluation Results

For evaluating our approach and measuring the effectiveness on an example collected dataset, we answer two questions:

· **Q1:** How effective is SEMFLOW in detecting benign activities and dealing with non-related ones?
· **Q2:** What is the performance cost of SEMFLOW?

To answer these questions, we first need to generate signatures for each behavior in each application (Table 2). Then, we compare with the ground truth which are trace files containing the information about when an operation was taking place in a sample trace file. Afterwards, we compare the labeled events to finally extract the coverage of event labeling as well as the false positives.

5.1   Experiment Setup

Collecting Matching Behavior For selected behaviors, we first collect traces from behaviors that we *can* identify (from a user point of view), and we initially set to collect three datasets having different durations and interacting with different files (i.e., downloading file *A* then downloading file *B*). This ensures that we are not dependent on the objects involved. Instead, it confirms that we can detect the activity no matter the duration nor the involved objects might be. These three datasets are used to extract unique callstacks that may represent a certain activity, ensuring the uniqueness of events that are related to it. (We later discuss in **Accuracy** section about this number).

Collecting Non-Matching Behavior: The non-matching behavior is intended for filtering out non-unique events that we obtained from the previous step. In this step, we do anything but the selected activity to finally find out unique events that we believe are related only and only for performing our selected activities. Since non-matching traces may result from more activities, we collect more number of traces than the previous step; namely, five traces. This is because programs have lots of more functionalities than what we usually expect from them (i.e., their main functionality). For example, almost any program would have a "setting" menu that would generate events not related to our intended activity set. Although the mentioned behaviors usually do not generate too many events, they can help filter out the ones that would happen in different contexts, leading to a reduction in our signature size, as well as decreasing false positives.

Test Sets: After collecting unique events for activities, we generate mixed traces, meaning the traces that are collected from a normal user interaction with the program, not restricted to any specific behavior. For collecting ground truth (i.e., the parts of the trace that we are supposed to *know* what the user is doing, versus the rest of the program execution), we ask the users to trigger a button and specify the activity they will be doing from the list of our detected activities in Table 2. Then we record the timestamp ranges where the specified operations were done. Hence, the rest of the trace would be the part that we have done anything *but* the said operations. The purpose of timestamp recording is to verify that none of the events that we have identified their operation, appear in the parts that we are not doing any of those that we have detected in Table 2. In test sets, we try to cover more activities and more running time than the mentioned ones in Table 3, since they were already used to generate the non-matching traces. But the actual activity might not be high-level enough to be put in Table 3, because some activities (such as background ones) overlap the ones put in the mentioned table. For example, in MS Word, we collect changing layout to a certain size and double-column the page, while in test sets, we also perform a change of indentation and more available actions in the layout tab.

Evaluation Methodology Our system has been evaluated both in terms of accuracy and performance. In this section, we discuss how the generated signatures perform in a real-world system, and how we assess the overhead it suffers in labeling events.

Accuracy: For evaluating our signatures, one general consideration is why we chose numbers of three for matching and five for non-matching behavior to generate signatures. The reason is, based on experiment results, if we choose the minimum of having one matching and one non-matching trace, we would have more false positives since we cannot accurately filter unrelated events. This is considered to be a general concept in any learning-based solution; if one has more training data, the more accurate their results would be. Future researchers may choose to use more or less numbers, according to their performance/accuracy requirements.

– **False Positives:** Denote percentage of events that we *do* label, but are outside the timestamps of which the user claims performing a known activity (i.e., we mislabel them).
– **True Positives:** This rating will indicate the percentage of events that are solely related to an activity, which are labeled as they should be. Evaluating related events would require ground truth about the program implementation, which is often difficult to denote since most programs are closed-source, and even the open-source ones are often too complicated to analyze. Hence, we manually extract events that we are confident to be related to a particular activity and check how many of them we were able to identify. In order for that, we manually analyze signatures and the activity we are monitoring and extract events in the traces that correspond to the files involved in that activity. An example of such system knowledge would be in extracting events from a WinSCP file transfer. A little system knowledge can show us that when transferring files, the WinSCP program would not load the whole file at once in memory when sending it out. Hence, we expect the `ReadFile` event reading the source file to be one of the most frequent events in our traces. Then we see if we have included the callstack signature for that event in our traces. We can also perform a backward check by recording the objects involved in behaviors. As an example, for the Firefox file download signature, we also record the file paths of downloaded files, and we ensure that event callstacks in our signatures match the filename as the same object because we expect no other event in the trace to have the same filename and callstack in the trace, except the ones that were associated with the file download behavior.

Performance: Signature generation uses the simple filtering technique described in Algorithm 1, and the overhead essentially depends on the size of the trace files being collected. For deployment of the embedded collector and matcher, we used the normal ETW collector to perform collection of unlabeled events, then the comparison is done by adding the callstack parsing and event labeling part. Our system needs to iterate through each frame in the callstack, ranging between 10-69 frames in our datasets; then, for each frame, we find the related library according to the address we receive. This information is extracted from the *LoadLibrary* events (which are normally collected by ETW), providing the library load address for each loaded library on the system. We then determine if the library is under *C:/Windows* (except for MS-developed apps), then

we find if the detected library is a user-level library. At last, we extract a hash out of the recorded addresses in the callstack for each event, adding it as an additional parameter in the event, which will be later compared to the signatures (if any) extracted from each activity.

5.2   Results

Table 3 shows the results for FPs and TPs. For some, we do have some FP since the same event would be used in a different context as well, such as in MS word where the complexity of the application would result in some non-deterministic events that would randomly occur in matching traces whereas, in the final test set, they would also appear in parts where the behavior was not taking place.

False Positives: In some cases, we have a few events that are considered to be unique as given in the matching traces and would not appear in the non-matching traces too. But, in test sets, we still see them occurring in the non-matching parts of the test trace. Hence, they are considered to be false positives. Analyzing the reason of why we have these FPs is a bit challenging since we do not have the source code for most applications, and for the ones we do, analysis of the code and following the code structure by a human is rather frustrating. One case for our false positive analysis is the false positive scenarios for Microsoft word. After analyzing the false positives, we look into the file paths of the files being read. It turns out two of the false positives read a file related to proofing data, the other four, read an `index.dat` which is also used by other Microsoft titles. There is no official documentation about the purpose of this file, but seemingly, this file is used to maintain file links in Microsoft applications and is read for multiple purposes. Hence, we can get the conclusion that some complex applications, perform different tasks in different scenarios and they decide how to perform the task in realtime, and it might not be deterministic for the application

Table 3. List of mixed operations used for testing SEMFLOW

Application	Operations	Duration (m)	Event Size	FPR	TPR
VLC	Video Playing, Theme Change, Check Updates	50	4250	0%	100%
FileZilla	File Upload & Download	20	90K	0%	100%
WinSCP	File Upload/Download/Copy (local), File Browsing, Adding Key, Changing Storage	20	50K	0%	100%
Notepad	File Save, Change Font, Print, Change Page Setup	15	14K	0%	100%
Sublime Text	File Save, Sorting, Setup Developer Commands, Change Color Scheme	15	25K	0%	100%
Skype	File Upload/Download, Upload Profile Photo, Change Settings, Add Contacts	25	120K	0%	100%
WinRAR	File Compression, Browsing, Test and Repair Archives, Licensing, Check for Updates	10	230K	0%	100%
Firefox	File Upload/Download, Install Extension, Change Settings, Browse History, Use Developer Tools	15	3M	5.6e-6%	85.67%
Chrome			7M	0.1e-3%	91.2%
MS Edge			4.5M	0.0013%	100%
Cobiansoft Cobian Backup	Routine Backup, Check Logs, Enc/Decrypt, Set Permissions, Edit Tasks	40	713K	0%	100%
Outlook	File Upload, Email w/o Attachments, Change Panes, Browse Folders	15	80K	0.1e-4%	100%
Windows Explorer	File Copy/Move, Browse, Search, Edit Taskbar, Create/Delete Objects, Change Properties	15	220K	0%	100%
Microsoft Word	Change Page Layout, Settings, Type and Save, Open New Doc	30	190K	0.078%	100%
TeamViewer	Connect, Remote Control, File Transfer, Local Settings	20	195K	0%	100%
Adobe Reader DC	Open and Fill Forms, Change Window Layout	10	20K	0.0015%	100%

Table 4. FP Rate for Different Number of Training Sets

Activity	# Matching	# Non-Matching	FPR
VLC Video Play	1	1	0.1e-4%
	1	3	0%
	2	3	0%
MS Word file manipulation	1	1	18.6%
	1	3	4.44%
	2	3	0.08%
Firefox Download	1	1	1.8%
	1	3	0.004%
	2	3	0.1e-3%
Explorer File Copy	1	1	1.8%
	1	3	0.03%
	2	3	0.1e-5%

to behave the same in two different situations, while it is seen by the user to be performing the same task; as in MS word, typing and saving.

True Positives: For true positives, as mentioned in 5.1, we identify events that we expect to happen in a sample behavior. Results show that have included almost all related events from a user point of view, in the signatures. The only event that we had expected to see and we did not, was the final file rename of the *.part* file to the actual file name in Firefox. Our analysis shows that Firefox performs the file rename in a JavaScript submodule. And as we discuss in Section 7, interpreted languages' callstacks cannot be used to identify an activity as the code path will be determined by the interpreter at runtime.

The performance overhead of collecting callstacks in different applications, is measured by measuring the percentage of the CPU core which is collecting ETW events, compared to when we are only recording events and not collecting callstacks not labeling them. It might be also worth mentioning that both the collection and labeling take place in one thread; as the overhead shows, we do not need to dedicate more CPU resources to it. Our analysis show that in the most complicated application which generates the most events (file download in Firefox), the overhead remains less than 3.5 percent; showing much lower overheads for less complicated applications.

As for signature generation, since it does not need to be done on the user endpoint, we just show that it is also even feasible in a common machine. We used a desktop Intel Core i7 @3.60GHz machine to generate signatures. The duration essentially is connected to how many events we have in our matching and non-matching traces, and how many traces we collect from each activity, and it increases linearly according to our implementation of Algorithm 1. The results in the sample of Firefox download signature show 6 minutes of generation time. Whereas, in the simple case of VLC, it only takes 8 seconds and in the average of MS word file manipulation, it takes 1:30 minutes.

Table 4 also shows some of the FP rates for having different numbers of matching and non-matching traces. The results are taken from the best choice of the said traces. As a general learning concept, the more training data, results in the observation of more accuracy rates.

5.3   Signature Verification

Upon collecting signatures from different activities, it is worthy to check if the extracted signatures represent the behavior that they were extracted from. It
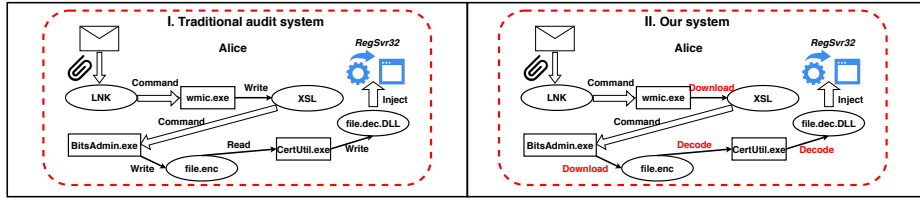
Fig. 5. Attack scenario in 6 without (left) and with (right) semantic identification mainly remains a challenging task to perform the given task; given that most of the analysed programs are closed-source, and the open-source ones may be difficult to be analysed by human in order to verify the callstacks that appear in our signatures. In order to provide a general methodology for security analysts to verify the signatures, we provide a guideline on how we evaluate the accuracy of the signatures. The operations we have analysed, often have multiple objects involved within. For each application, we check whether the callstack of the related object does appear in the signature. Such as in Table 1, we can see the downloaded file name, the temporary file used during the operation, and the SQLite operation which adds the downloaded file to downloads library in Firefox.

Hence, we are able to verify signatures similar to how we calculate TPs mentioned in Section 5.1, so that security analysts would be able to refine the signatures generated from SemFlow; and revise them, according to their findings.

The need for manual involvement cannot be ignored, as at least the first time of signature generation from a software needs supervision to see if the signature and the involved objects in the event can actually represent the behavior and not just irrelevant events. The expectation is that minor updates to programs will not affect the sequence a program follows to generate the signature. Hence, after detecting the type of the behavior and verifying if the events in the signature are reasonable, the generation can happen automatically for future versions of the same program by replicating the same user behavior used to generate signatures from the original software version.

6   Case Study

In this section, we analyse a Living-off-the-Land attack, in which parts of the attack use existing tools on the system to finally perform a malicious DLL injection onto the system [6]. Figure 5 shows the attack example with the right graph being labeled by our system. An initial email contains a LNK file that uses `wmic.exe` to download XSL files. The downloaded file contains scripts for `BitsAdmin.exe` to download an encoded file. Also, the encoded file is a malicious DLL which is finally injected onto the system using the `RegSvr32` tool. As seen, our system can label the LotL operations being performed during the attack for lateral movement. Tools such as `wmic.exe` are also used for LotL purposes and we can easily identify the operations done by such pre-installed applications by generating signatures from specific operations.

7   Limitations

While being effective, our system has certain limitations. Our signatures rely on program implementation, which leads to a few problems. Firstly, if the code of a program changes, the signature also needs to be updated. We can easily overcome this limitation by recording the activities once and performing them again automatically using automation tools, collecting matching and non-matching behaviors from updated versions of applications.

Secondly, we cannot deal with cases where interpreters are used (e.g., Python), because in this case, the code path is decided by the interpreter at runtime, changing the callstack according to the interpreter's decisions. However, since the code is visible to interpreters, it would also be visible to a system trying to extract semantics from the code, or in Java, from the bytecode; making efforts possible for performing static analysis, which can be a future direction for researchers.

Another limitation is receiving accurate callstacks for network events. In our experiments, all network-related events have the same callstack, all frames pointing out to the kernel libraries. Hence, in addition to control flow operations, only file-related data flow operations can be labeled, so we completely ignore the labeling of network-related events as ETW cannot provide application-level callstacks for these types of events.

Lastly, our current implementation does not consider the connection of traces that represent the execution of a behavior and its dependencies. For example, the Spotify music playing application first caches a few songs when it starts and reads the cached songs from memory if the user decides to play them. In this case, SEMFLOW fails to get to know the caching behavior because it happens in the starting phase and is not unique for playing. And most importantly, it cannot be triggered by the user in order to collect traces specific to that operation.

8   Related Work

Callstack. SEMFLOW uses callstack information in order to extract semantic information from low-level system events. Using callstack information is used in other contexts as well. IntroPerf [19] uses such information to rank performance bugs using ETW. PerfGuard [18] also uses callstacks to apply performance assertions on application transactions.

Detection. There have been works on provenance analysis and attack reconstruction [12, 29, 26], which have proven to be accurate and efficient. However, not using callstack information from events may mislead those works as insider attacks tend to use similar events in benign scenarios. Our work was able to label individual events using callstack. We believe that even without semantic extraction, this data can be largely used to distinguish between events, resulting in a more accurate attack graph, as our semantics labels can be integrated to and improve those works.

Some works also focus on detecting system call sequences and marking attack sequences [14, 25]. While the system call data has been shown to be more fine-grained, they would generate more events with less to be extracted from, compared to event-level work. [31] uses system call arguments to mitigate this

Table 5. Comparison of Semantic Identification Works

Work	Training Overhead	Accuracy	Detection Overhead	Dependency on Source Code
OmegaLog	High (instrumentation)	High	Low	Highly-Dependent
MPI	Low	High	Low	Highly-Dependent
MCI	High (instrumentation)	High	Low	None
UIScope	Low	Low	Low	None
SemFlow	Low	High	Low	None

problem, but ETW does not provide arguments for individual system calls, yet it can provide the arguments for system-level events instead.

Machine-learning work tend to train data based on previous attack and benign scenarios [16, 36], but they have been shown to have many false-positives and a huge reliance on having enormous training data for accuracy increment [10], in addition to being hard-to-verify.

CONAN detects APT attacks by translating the top kernel callstack address to its API call and contracting a chain of Windows kernel API calls to detect malicious behavior [37].

Data Reduction. There also have been efforts to reduce the amount of data [24, 38, 32], which will be stored into permanent storage for future analysis. These works focus on maintaining provenance information between objects; so that the security analysts would be able to analyze the attack, even with reduced number of events. By using semantic information, in some cases, provenance information can be ignored or aggregated to save more space; if we completely are certain that an event cannot play any role in an attack scenario. Hence, we claim that by using semantic information, some events and their related objects can be ignored and the dependency information is not even needed in "harmlessly-labeled" events. Such an example would be the semantics of "VLC Video Play".

Provenance Tracking. Execution partitioning works such as [35, 23] require invasive training and instrumentation tools to perform provenance tracking, while our methodology uses the same data collected on the user machine to perform event labeling.

SEMFLOW generally is not an independent provenance constructor, and it only performs event labeling for future graph extraction using existing methods. Some use *semantic detection* [28, 22, 21] along with binary analysis for causality inference. But, binary analysis tends to be expensive while our approach is quite simple, and it uses the same type of data collected from the user machine to label the realtime event flow, making the process more clear to security analysts in each step. Some other use *tainting*[27, 17, 2, 30, 7], which generally is an expensive task and hence cannot be performed in realtime. There have been works towards optimizing it using hardware modifications [34, 33] but this is impractical, especially when using built-in data collectors in mostly common user systems which use Windows as their main operating system without any hardware modification or kernel access.

High-level Semantic Identification. Table 5 shows an overview of the existing semantic identification works. Works such as [28, 22, 13, 39] are focused on extracting higher-level representations of low-level system data in order to make intrusion analysis easier for security experts. However, each suffers from multiple flaws. MPI [28] requires annotating high-level structures in source code.

However, requiring source code is impractical on the Windows platform as most Windows internal and third-party applications are closed-sourced. Authors of MCI [22] use LDX [21] to extract causality information from system calls. However, in Windows, system calls do not provide their arguments individually, and only for some of them, we can extract the corresponding parameter by leveraging system events. Also, it requires cumbersome training, and in comparison, it has higher false positives. Besides, model matching is not done at runtime as events arrive into the stream, and is performed after an attack is detected. It is also worth mentioning that system calls occur many times more than system-level events since they provide a lower-level abstraction of the system. It also faces the same limitation of needing to regenerate models upon receiving a newer version of the same program binary. Some works provide high-level contexts by leveraging application logs. OmegaLog [13] analyzes program binaries to identify and model application-layer logging behaviors, enabling accurate reconciliation of application events with system-layer accesses. lprof [41] and Stitch [40] observe that programmers will output sufficient information to logs so as to be able to reconstruct runtime execution flows. Based on the observation, they analyze application logs to provide semantic contexts for a single request without instrumenting any distributed application. However, all those works highly rely on the quality of application logs. It means that those works cannot work on applications which do not provide rich and sufficient application logs. Furthermore, lprof and OmegaLog require binary static analysis which is not available for binary codes built dynamically at runtime. In contrast, our system provides a general way based on general system events to recover high-level application behaviors without binary analysis. UIScope [39] correlates low-level system events with high-level UI events to provide high visibility. However, UIScope can only work on GUI applications while our system can work on both GUI and non-GUI applications. On the other hand, some works [29, 11] map low-level system events to high-level Tactics, Techniques, and Procedures (TTPs) and connect them on generated provenance graphs to accelerate attack investigation. Those works leverage domain knowledge to generate rules to do mapping. In contrast, our work provides a general and accurate way to automatically identify high-level behaviors, which can supplement those works.

9   Conclusion

We develop SEMFLOW, which uses system-level data that is manually labeled by the activity that was run, then finds events that are responsible and used in a specific activity. Then let the ETW collector know about the label of each event it receives in realtime. The overhead for labeling is negligible as the collector can use set matching to perform labeling on individual events. We then implemented a prototype of the system and showed that it can expand and improve the robustness of existing work in detection and forensics by being able to distinguish between events.

References

1. Hardening Windows 10 with zero-day exploit mitigations. https://bit.ly/2KdiTiv (2017), accessed on 2017-6-10
2. Taintgrind. https://github. com/wmkhoo/taintgrind (2017), accessed on 2017-12-10
3. Windows-10-Mitigation-Improvement. https://ubm.io/2IIVwtn (2018), accessed on 2018-4-10
4. APT1. https://bit.ly/2D7RNHI (2019), accessed on 2019-5-4
5. Living off the Land: Attackers Leverage Legitimate Tools for Malicious Ends. https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/living-land-legitimate-tools-malicious (2020), accessed on 2020-10-10
6. Living off the Land: Turning Your Infrastructure Against You. https://docs.broadcom.com/doc/living-off-the-land-turning-your-infrastructure-against-you-en (2020), accessed on 2020-10-10
7. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: OSDI. vol. 10, pp. 1–14 (2010)
8. Gao, P., Xiao, X., Li, D., Li, Z., Jee, K., Wu, Z., Kim, C.H., Kulkarni, S.R., Mittal, P.: SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In: USENIX Security (2018)
9. Gao, P., Xiao, X., Li, Z., Xu, F., Kulkarni, S.R., Mittal, P.: AIQL: Enabling Efficient Attack Investigation from System Monitoring Data. In: USENIX ATC (2018)
10. Harang, R., Kott, A.: Burstiness of intrusion detection process: Empirical evidence and a modeling approach. IEEE Transactions on Information Forensics and Security **12**(10), 2348–2359 (2017)
11. Hassan, W.U., Bates, A., Marino, D.: Tactical provenance analysis for endpoint detection and response systems. In: Proceedings of the IEEE Symposium on Security and Privacy (2020)
12. Hassan, W.U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., Bates, A.: Nodoze: Combatting threat alert fatigue with automated provenance triage. In: NDSS (2019)
13. Hassan, W.U., Noureddine, M.A., Datta, P., Bates, A.: Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In: Proc. NDSS (2020)
14. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of computer security **6**(3), 151–180 (1998)
15. Hossain, M.N., Milajerdi, S.M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., Venkatakrishnan, V.: {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 487–504 (2017)
16. Hu, W., Liao, Y., Vemuri, V.R.: Robust anomaly detection using support vector machines. In: Proceedings of the international conference on machine learning. pp. 282–289. Citeseer (2003)
17. Jee, K., Portokalidis, G., Kemerlis, V.P., Ghosh, S., August, D.I., Keromytis, A.D.: A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: NDSS (2012)
18. Kim, C.H., Rhee, J., Lee, K.H., Zhang, X., Xu, D.: Perfguard: binary-centric application performance monitoring in production environments. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 595–606 (2016)
19. Kim, C.H., Rhee, J., Zhang, H., Arora, N., Jiang, G., Zhang, X., Xu, D.: Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. ACM SIGMETRICS Performance Evaluation Review **42**(1), 235–247 (2014)

20. King, S.T., Chen, P.M.: Backtracking intrusions. ACM Transactions on Computer Systems (TOCS) **23**(1), 51–76 (2005)
21. Kwon, Y., Kim, D., Sumner, W.N., Kim, K., Saltaformaggio, B., Zhang, X., Xu, D.: Ldx: Causality inference by lightweight dual execution. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 503–515 (2016)
22. Kwon, Y., Wang, F., Wang, W., Lee, K.H., Lee, W.C., Ma, S., Zhang, X., Xu, D., Jha, S., Ciocarlie, G.F., et al.: Mci: Modeling-based causality inference in audit logging for attack investigation. In: NDSS (2018)
23. Lee, K.H., Zhang, X., Xu, D.: High accuracy attack provenance via binary-based execution partition. In: NDSS (2013)
24. Lee, K.H., Zhang, X., Xu, D.: Loggc: garbage collecting audit log. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1005–1016 (2013)
25. Lee, W., Stolfo, S.: Data mining approaches for intrusion detection (1998)
26. Liu, Y., Zhang, M., Li, D., Jee, K., Li, Z., Wu, Z., Rhee, J., Mittal, P.: Towards a timely causality analysis for enterprise security. In: NDSS (2018)
27. Ma, S., Lee, K.H., Kim, C.H., Rhee, J., Zhang, X., Xu, D.: Accurate, low cost and instrumentation-free security audit logging for windows. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 401–410 (2015)
28. Ma, S., Zhai, J., Wang, F., Lee, K.H., Zhang, X., Xu, D.: {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1111–1128 (2017)
29. Milajerdi, S.M., Gjomemo, R., Eshete, B., Sekar, R., Venkatakrishnan, V.: Holmes: real-time apt detection through correlation of suspicious information flows. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1137–1152. IEEE (2019)
30. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security. pp. 1–25. Springer (2008)
31. Tandon, G., Chan, P.K.: On the learning of system call attributes for host-based anomaly detection. International Journal on Artificial Intelligence Tools **15**(06), 875–892 (2006)
32. Tang, Y., Li, D., Li, Z., Zhang, M., Jee, K., Xiao, X., Wu, Z., Rhee, J., Xu, F., Li, Q.: Nodemerge: template based efficient data reduction for big-data causality analysis. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1324–1337 (2018)
33. Tiwari, M., Li, X., Wassel, H.M., Chong, F.T., Sherwood, T.: Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 493–504 (2009)
34. Tiwari, M., Wassel, H.M., Mazloom, B., Mysore, S., Chong, F.T., Sherwood, T.: Complete information flow tracking from the gates up. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. pp. 109–120 (2009)
35. Wang, F., Kwon, Y., Ma, S., Zhang, X., Xu, D.: Lprov: Practical library-aware provenance tracing. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 605–617 (2018)
36. Wu, J., Peng, D., Li, Z., Zhao, L., Ling, H.: Network intrusion detection based on a general regression neural network optimized by an improved artificial immune algorithm. PloS one **10**(3), e0120976 (2015)

37. Xiong, C., Zhu, T., Dong, W., Ruan, L., Yang, R., Chen, Y., Cheng, Y., Cheng, S., Chen, X.: Conan: A practical real-time apt detection system with high accuracy and efficiency. IEEE Transactions on Dependable and Secure Computing (2020)
38. Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., Jiang, G.: High fidelity data reduction for big data security dependency analyses. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 504–516 (2016)
39. Yang, R., Ma, S., Xu, H., Zhang, X., Chen, Y.: Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In: Network and Distributed Systems Symposium (2020)
40. Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., Stumm, M.: Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp. 603–618 (2016)
41. Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D., Stumm, M.: lprof: A non-intrusive request flow profiler for distributed systems. In: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). pp. 629–644 (2014)