CONAN: A Practical Real-time APT Detection System with High Accuracy and Efficiency

Chunlin Xiong, Tiantian Zhu, Weihao Dong, Linqi Ruan, Runqing Yang, Yueqiang Cheng, Yan Chen, *Fellow, IEEE,* Shuai Cheng, and Xutong Chen

Abstract—Advanced Persistent Threat (APT) attacks have caused serious security threats and financial losses worldwide. Various real-time detection mechanisms that combine context information and provenance graphs have been proposed to defend against APT attacks. However, existing real-time APT detection mechanisms suffer from accuracy and efficiency issues due to inaccurate detection models and the growing size of provenance graphs. To address the accuracy issue, we propose a novel and accurate APT detection model that removes unnecessary phases and focuses on the remaining ones with improved definitions. To address the efficiency issue, we propose a state-based framework in which events are consumed as streams and each entity is represented in an FSA-like structure without storing historic data. Additionally, we reconstruct attack scenarios by storing just one in a thousand events in a database. Finally, we implement our design, called CONAN, on Windows and conduct comprehensive experiments under real-world scenarios to show that CONAN can accurately and efficiently detect all attacks within our evaluation. The memory usage and CPU efficiency of CONAN remain constant over time (1-10 MB of memory and hundreds of times faster than data generation), making CONAN a practical design for detecting both known and unknown APT attacks in real-world scenarios.

Index Terms—Advanced Persistent Threat, Hosted-based Security, Real-time Detection,

1 INTRODUCTION

HOST security has been studied by researchers for decades. The evolution of attacks has rendered existing approaches insufficient for modern sophisticated scenarios. Advanced Persistent Threat (APT) attacks are used by many experienced attackers to compromise victims' hosts due to the persistence, stealth, and clear goals of these attacks. APT attacks are usually initiated by hacker groups with a national or organizational background. Hence, these groups are well-organized, well-targeted, highly skilled, and highly invasive. The 2019 annual report from FireEye showed that more than twenty active APT groups had launched attacks against targets in dozens of domains, including governments, financial companies and even the Winter Olympics [1].

Traditional intrusion detection methods can be divided into two categories: offline and online. One of the most famous offline detection methods is the sandbox approach, where the target program is deployed to an isolated environment for separate analysis [2]. In addition, several logging and provenance tracking systems have been built to monitor the activities of systems and then build provenance graphs to detect or analyze attacks [3–11]. Although these methods allow a clear view of the attacks, considering the hysteresis of offline detection, people have begun using online detection methods to detect attacks in real-time. These approaches include network traffic-based analysis [12–14], software static feature detection [15, 16], and hook technology [17, 18], among others. However, existing research has focused mainly on one specific stage of APTs and the intrinsic mechanisms and attack vectors of APTs remain poorly understood.

Context-based detection has been proven to be effective in recent works [19]. Real-time detection systems with contextual methods have been proposed in recent years. StreamSpot [20] analyzed streaming information flow graphs to detect anomalous activities by extracting local graph features and vectorizing them for classification. Learning-based detection methods can only provide malicious scores or classification results but cannot explain these results. Furthermore, these detection systems cannot detect attacks without false alarms. Therefore, in practice, learning-based methods are unsuitable in enterprise scenarios. Sleuth [21] subsequently proposed a tag-based detection method based on provenance graphs, but this method focused mainly on suspicious access to confidential files and reduced false positives by adding a domain white list. To gain a better understanding of APT attacks, analysts have decoupled the APT life cycle into multiple phases and then used the corresponding features of each phase to match the suspicious behavior. APT attacks were divided into seven or eleven phases [1] [23–25]. The multiphase kill chain model approaches were adopted by numerous researchers.

- *C. Xiong, W. Dong, L. Ruan, R. Yang and S. Cheng are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: chunlinxiong94@zju.edu.cn, dwh@zju.edu.cn, ruanlinqi@zju.edu.cn, rainkin1993@zju.edu.cn, 21821303@zju.edu.cn.*
- *T. Zhu is with the College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China. E-mail: ttzhu@zjut.edu.cn.*
- *Y. Chen and X. Chen are with Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA. E-mail: ychen@northwestern.edu, xutongchen2021@u.northwestern.edu.*
- *Y. Cheng (the corresponding author) is a security scientist at Baidu Security. E-mail: chengyueqiang@baidu.com.*

1. There is one more phase, *Impact*, added in ATT&CK model after the publication of [22], it contains twelve phases now.

Holmes [22] achieved substantial progress in phase-based detection by building a model that detected each phase based on simple rules and computed suspicious scores. However, these phases are not all necessary in APT attacks (e.g., Credential Access), and some of them (e.g., detecting remote code execution vulnerability) are usually detected via a priori knowledge and tend to change over time.

Additionally, real-time contextual work [21, 22, 24] usually preserves context information in a *provenance graph* [4]. However, the graph continues to grow over time, and APTs can last for months or even years, making these approaches inevitably suffer from efficiency and memory problems when the system runs for long periods, especially for real-time detection [26]. As a result, most detection methods rely on short time windows [27–29].

To address these challenges, especially, **accuracy** and **efficiency**, this paper presents a model for accurately detecting APTs. Furthermore, it presents a novel state-based detection framework in which each process and file is represented as a fine-designed data structure for real-time, long-term detection.

To detect unknown APTs with high accuracy, instead of concentrating on unnecessary, undetectable and easy-to-change phases in APT attacks, we utilize control flow (i.e., why a process or code is being executed) and data flow (i.e., how data are passed among objects) to explain contextual behavior. We identify the following three essential attack phases: 1) deploy and execute the attacker's code, 2) collect sensitive information or cause damage, and 3) communicate with the C&C server or exfiltrate sensitive data. We focus primarily on accurately detecting these phases and combining them to distinguish malicious behaviors from benign ones. Compared to more complicated phase-based modules, this approach is beneficial for accurately detecting unknown APTs.

To conduct such contextual detection in real-time with high efficiency, we propose a novel state-based tracking and detection framework and a corresponding data structure based on ideas from forensic analysis. In this design, all semantics are stored as states, and the framework keeps only the current states of all processes and files for detection. States are updated by events and related states of other entities, similar to finite state automata (FSA), which we call it FSA-like structure. Consequently, the framework does not need to store historic data, and memory usage remains consistent. The states change over time. Once a process changes into a malicious state, the attack is detected no matter how long it lasts. Using this framework, we can monitor the host over long periods of time to automatically detect APTs with high accuracy and low overhead.

Moreover, our detection method based on this framework can detect attacks and provide explanations. Specifically, the detection results are generated with reconstructed attack graphs, which illustrates how these attacks happened and benefits subsequent analysis.

Finally, we implement our design, called CONAN[2]. CONAN collects data from Windows, extracts semantics, and then uses an intelligent strategy for state transfer and to detect APTs in real-time. Moreover, it can automatically reconstruct part of the attack chain once an attack is detected. We evaluate CONAN under three scenarios: DARPA engagement, our laboratory and three real-world companies. The results show that CONAN can detect all potential APT attacks in our experiments rapidly with near zero false positive rate and accurately reconstruct attack graphs. The memory usage of CONAN remains constant (1-10 MB) over time, unlike previous designs built on growing provenance graphs [21, 22]. We summarize our contributions as follows:

- We propose a novel model for APT detection that concentrates on three constant steps of APTs, and we present a set of designs to accurately track and detect these steps, including detecting memory-based attacks and suspicious process behaviors.
- We present a novel and efficient state-based detection framework, in which each process and file is represented as an FSA-like structure. This framework helps to detect APTs with constant and limited memory usage (1-10 MB) and high efficiency (hundreds of times faster than data generation).
- We implement our design as an APT attack detection system that can detect unknown advanced attacks in real-time with high accuracy and rapidly restore the attack chain. We tested our system on real-world datasets and determined that it performs better than previous approaches, especially in terms of efficiency and accuracy.

The atomic suspicious indicators (ASIs), transition rules and malicious states in this paper are the same as those used in evaluation. Although CONAN can detect all attacks presented in DARPA Engagement, we do not aim to prove that these definitions are enough for all attacks; instead, we want to present our design to detect and reconstruct APTs accurately and efficiently. Moreover, CONAN can easily be extended by adding more data sources, because ASIs, rules and malicious states are customizable in configuration files.

The remainder of this article is organized as follows. We first describe the threat model in Sec. 2. Then, we introduce the detection model and a novel detection framework in Sec. 3 and Sec. 4, respectively. We evaluate CONAN in Sec. 5. The discussion, related work and conclusion are presented in Sec. 6, Sec. 7 and Sec. 8, respectively.

2 A LIVING EXAMPLE AND THREAT MODEL

In this section, we give a simple example of APT attacks to illustrate the disadvantages of existing work. This attack begins with a phishing email because, as concluded by PhishMe research [30], 91% of the time, phishing emails are behind successful cyber-attacks.

Imagine you are working on your computer and receive an email from a colleague asking you to fill a table in an attachment. You download the attachment without receiving alerts from anti-virus software, open it, fill it in, then reply to the sender. You may forget about this email after a few days, but months later, you find that some sensitive data have been stolen by competitors.

Furthermore, the attack could be more sophisticated. For example, the attacker first prepares a new Remote Access Trojan (RAT) that is never discovered by anti-virus vendors.

2. CONAN for CONtext-based apt detection by Automatic provenance aNalysis.

Then, he collects information about your company so that he can emulate the way employees send you emails. The attachment contains an obfuscated macro script executed when the file is opened. It executes PowerShell commands to download and run the prepared RAT in memory, providing access to your machine without leaving malicious files on hard disks. The attacker can lurk for a long time until he gets what he wants, such as a sensitive file or the password to your accounts.

Here, network traffic-based analysis can be invalid due to camouflage and encryption. Sandbox-based detection may be passed by anti-sandbox techniques. Static malware analysis cannot work because there is no malware left on the disk and even the malware in memory is completely new with no static features recorded.

In our threat model, an attacker knows his target well and prepares a new attack including zero-day vulnerabilities, self-developed malware and a new DNS. He first makes his malicious code run on the victims' machines and then performs some malicious actions to collect information or cause damage, automatically or by accepting commands from the network. Finally, if he wants to obtain sensitive information, these stolen data should be sent via network. Although we deploy this system to detect APT attacks, it can also detect general attacks with similar goals. However, we cannot detect side-channel attacks and insider attacks, in which the attack has legal ways to access the machines. Another attack CONAN cannot detect is the return-oriented programming (ROP) attack [31]. With this technique, an attacker gains control of the call stack to hijack program control flow and then carefully executes chosen machine instruction sequences that are already present in the machine's memory. Thus, there is no malicious code deployed on the victims' machines. However, this attack can already be avoided or detected by existing methods [32–36].

In this paper, we assume that the event logs and digital signatures are credible. All attacks used in the evaluation cannot be detected by traditional anti-virus systems. In contrast, we do not need to assume the whole attack happens after the installation of our system. As a result, pre-installed malware and malicious code can be detected.

3 DETECTION MODEL

In this section, we first provide an overview of our design as shown in Fig. 1. We develop a fast and stable multiple-level data collector on the host to collect audit traces, call stacks and additional data. Then, these data are sent to a detection server, extracted as high-level semantics and stored in an in-memory structure as process and file states. Meanwhile, all event logs are processed to change the states of processes and files based on predefined rules. These events and states are stored in a database for later attack reconstruction. Whenever a process enters a malicious state, an alert is triggered along with a reconstructed attack graph.

The detection model we present here makes the system accurately detect APTs, and the state-based framework makes it possible to efficiently detect APTs. We will introduce these two parts in Sec. 3 and Sec. 4, respectively.

3.1 Motivation

The letter **A** in APT stands for the advanced techniques used in these attacks. Traditional detection systems including malware detection, vulnerability detection and threat intelligence, focus on only a single phase of the APT attack chain, and the attack techniques used in these phases can be changed easily; thus, it is easy for APT attackers to evade these traditional detection methods. MITRE ATT&CK [23] introduces an eleven-phase APT attack model to describe the tactics, techniques and procedures (TTPs) [37] used in APT attacks.

However, an overly complicated multiphase model can be used only to better understand APTs, not to detect them. For example, the authors of [22] developed a system for detecting techniques in each tactic (phase) and connected these phases by information flows (including data flows and control flows) as attack chains. In each attack chain, the more phases detected, the greater possibility of an attack. However, there are three main problems. First, there are hundreds of techniques used in each phase. This system must detect hundreds of techniques, which is hard to implement and can cause high detection overhead. Second, the detection points of traditional methods, e.g., vulnerabilities, are still considered. To detect these phases, prior knowledge is required. Furthermore, the techniques used by attacks are prone to change; thus, it is hard to detect unknown attacks. Finally, the authors believe that although they cannot detect all phases accurately, as a subset of all phases, the detected ones are enough to distinguish attacks from benign activities. In other words, it is not necessary to detect all phases. Moreover, introducing some common and unnecessary phases would increase the false alarm rate.

More phases cannot indicate an attack, and less phases cannot prove its legality. For example, a new installing browser can trigger 6 phases in the MITRE ATT&CK eleven-phase model (Initial Access, Execution, Persistence, Credential Access, Discovery and Exfiltration), and will trigger a false alert by [22]. Meanwhile, an advanced attacker accesses the machine by a zero-day vulnerability and downloads malware. Then, it achieves persistence using an unknown method (or it does not want persistence in some scenarios, for example, on a never-shutdown server). It records keystrokes and exfiltrates the data over a command and control channel, which is probably hard to detect. In the end, the only stages that can be detected in this attack are Execution and Collection, so they cannot be considered as an attack by [22].

3.2 Three-phase Detection Model

To detect unknown APT attacks, we first find their invariant parts. In other words, we try to answer the following question: what makes these activities "attacks". After researching hundreds of APT attacks [38], it can first be observed that attackers must first deploy their code to victims. The difference is that malware may be customized or executed only in memory to evade traditional static file-based detection systems. The second observation is that the final targets of attackers remain the same over the years. APT1 [39] introduced attacks to steal hundreds of terabytes of data from at least 141 organizations since 2006. Today,

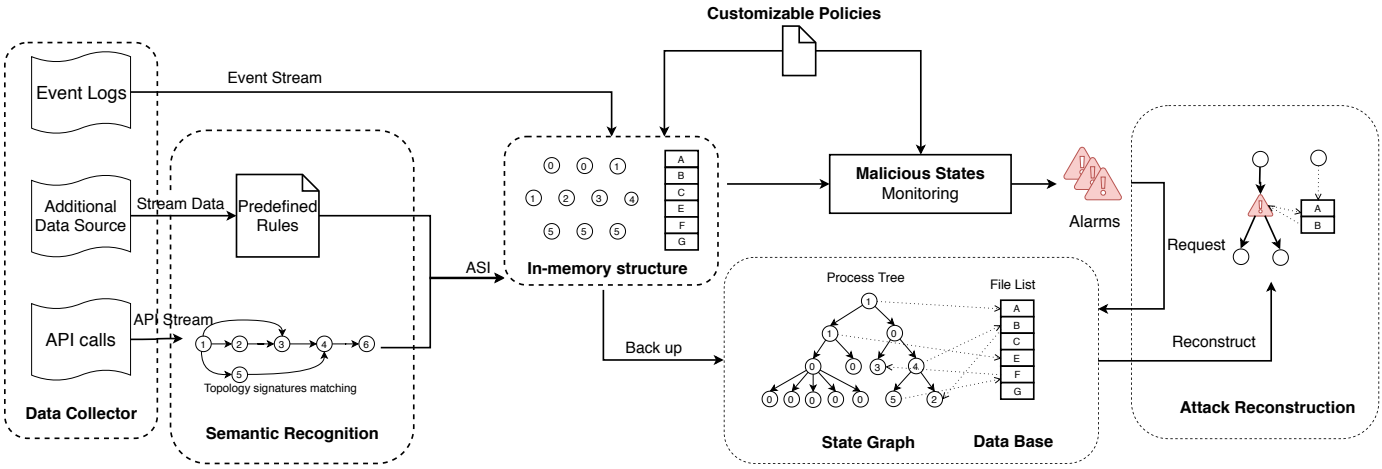


Fig. 1. Overview of CONAN. The data collector is deployed on the client side, and it tries to collect traces from multi-layers with low overhead. These traces are preprocessed on the client side to reduce the data size, and then sent to the detector on the server side. The detector constructs a main-memory structure to maintain the states of processes and files by predefined rules. States and necessary events are synchronized to the database. Once an alert is raised, the attack chain can be reconstructed from the database.

APT38 [40] also focuses on similar tasks. These behaviors are analogous to the permissions [41] in Android that may lead to privacy leakage or damage. The final observation is that attackers will communicate with C&C servers and exfiltrate confidential data, and the malicious program should always have the ability to access the network.

To this end, we propose a three-phase model to detect APTs:

- **Deploy and execute the attacker's code.** Any process behaviors are the outcome of code execution. The attacker must first deploy code to the victim to achieve his goal. To detect this phase, we monitor data flows from outside, including the network and portable devices, which we call *Untrusted Data Flows*. Having *Untrusted Data Flows* is the necessary condition for launching attacks, regardless any vulnerabilities or techniques the attacker uses to deploy his code. This design may lead to more false alarms, but the technique introduced in subsequent sections will help to resolve this problem. In another scenario, an attacker may use a legitimate process to achieve his purpose. For example, the attacker can capture screens by using the pre-installed Windows SnappingTool. Here, *Untrusted Control Flows* can help. If a process or thread is started by a suspicious thread, the process or thread is also marked as suspicious. Code deployment is always necessary unless the attacker can gain authorized access to the victim in some other ways(e.g., by logging in remotely with a password). This type of attack can be prevented by an IP white list and detected by anomaly detection, and it is outside the scope of our research.
- **Collect sensitive information or cause damage.** An attacker usually attempts to steal confidential data or to damage the victim's data or machines, which is the reason why the attacker implements the attack and why victims want to avoid such results. We do not consider intrusions that gain access to the victim but result in no harmful behaviors as real attacks. Stealing confidential data from files can be detected by monitoring data flows from confidential targets as *Confidential Data Flows*. Other suspicious behaviors are detected by our predefined signatures, as discussed in Sec. 4.1.
- **Communicate with the C&C server or exfiltrate sensitive data.** Both operations are necessary in APT. Without these operations, an attack cannot be accomplished. Although there are many ways to achieve this (e.g., removable disks), the typical real-world attack approach is through network connections.

These three phases are straightforward and necessary for most APT attacks. Thus, we try to detect the processes that *execute suspicious code to do malicious behaviors.* Moreover, we present additional features to illustrate different attack scenarios. Unlike previous work [21], we do not assign scores or simple tags; instead, we use more detailed descriptions to depict the components of attacks for better understanding and further analysis without additional overhead.

3.3 Tracking Suspicious Code Execution

Memory-based attacks, including injection and fileless attacks, which can help attackers execute their code in the memory of benign applications, are increasingly used because traditional detection systems are blind to them [42, 43]. While existing studies [20–22] regard a process as an entity for storing contextual information, they are vulnerable to memory-based attacks. Detecting techniques used in memory-based attacks could be helpful, but there are multiple ways to implement these attacks [44–46], which makes detection difficult. Thus, in this section, we propose a method for detecting suspicious code execution by tracking suspicious data flows and checking the execution call stacks ignoring the techniques attacks use.

If an attacker wants to execute malicious behaviors, he must 1) execute malicious code directly or 2) achieve it with the help of benign processes. The main challenges for detection are to determine 1) which code is executed, 2) where it comes from and 3) how it is executed. Although taint tracking helps a lot when tracking fine-grained data flows, this method cannot be used in real-time due to its

high overhead. To address the first challenge, we adopt call stacks. A call stack is a stack data structure that stores information about the active subroutine when an event is generated. The addresses in a call stack are return addresses belonging to different code blocks (e.g., images). If all addresses in one call stack come from trusted code blocks, we call this thread *executes trusted code*. Since benign processes can be easily forced to execute external code, we separate a process into multiple subunits (threads) based on code execution. As shown in Fig. 2, these threads that execute unknown or suspicious codes are separated from those that are totally benign. We consider the following scenarios:

Image Load and Memory Execution. Image load is a basic operation in which a process loads an executable file into its memory. An attacker can replace benign image files with malicious ones or force a benign application to load a malicious image and then conduct the attack under the guise of the benign process. Moreover, an attacker can write malicious code directly into another process's memory space or load an image from the memory instead of from the disk without triggering system events. The former is known as process injection, which is commonly used in attacks, and the latter is a technique, called reflective loading [45], which has been used in recent advanced attacks. Our system monitors the dynamic events for loading images and stores the base addresses and sizes of the allocated memory for each process. When the memory addresses of unsigned images or allocated memory appear in a call stack, it means that some unknown codes have been executed in that thread; thus, this thread should be separated from others. To reduce the overhead when parsing full call stacks, we conduct a sample inspection with a low frequency for each thread.

Script Execution. Script-based attacks have become common in recent years because the host processes are totally benign and the process reads the execution code, such as PowerShell and VBscript, instead of loading it. Detecting such attacks is challenging. Our system enumerates most of the common script hosts and treats them separately from normal processes.

To address the second challenge, we track the suspicious data by inference based on coarse-grained data flows. For example, if a process has a network connection, then any files written by this process could contain data from the network. These inference rules will be introduced in Sec. 4.3. Coarse-grained data flows may lead to a high false-positive rate, but the true-positive rate remains the same. The result in Sec. 5 shows that, even with coarse-grained data flows, our system can detect APTs with a low false-positive rate.

To address the third challenge, we track control flows (specifically, the processes created by suspicious threads). Although they may be benign processes, they can be used to achieve the attacker's goals, such as grabbing screens and exfiltrate sensitive data.

4 STATE-BASED FRAMEWORK

The letter **P** in APT stands for Persistence, which means the attacker can lurk for a long time until he gets what he wants. This is different from the same word introduced in attack chain models[23], which represents the techniques

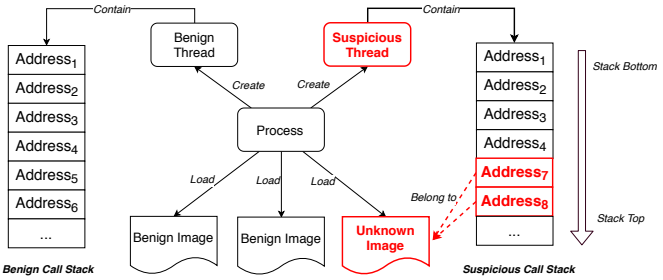


Fig. 2. A suspicious thread (red) that executes unsigned codes (red) is separated from the totally benign ones. This feature is important for detecting suspicious code execution, including unsigned images and in-memory attacks.

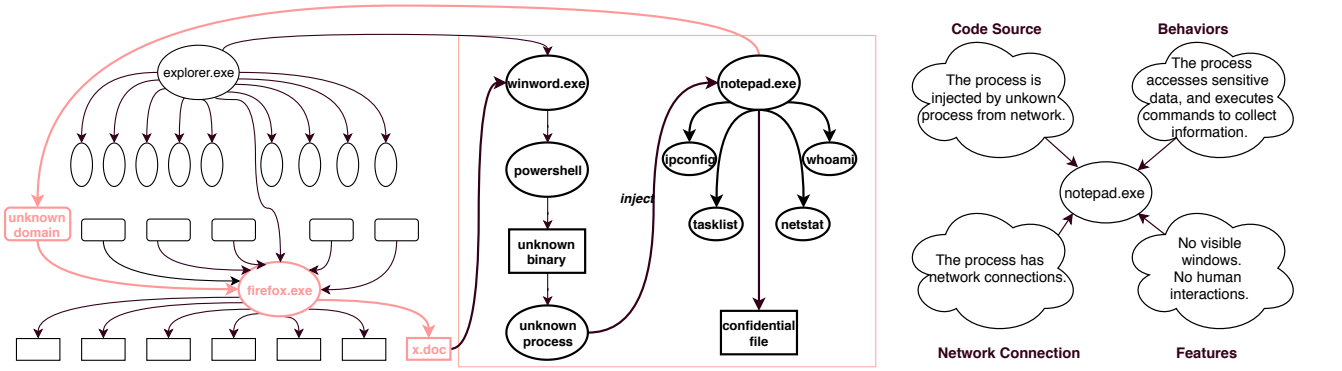
to make malware automatically start after the operating system restarts. Detecting the techniques of persistence is not practical. On the one hand, there are 59 known persistence techniques [23] and detecting them all could be expensive. On the other hand, even if a process is detected to be persistent, it cannot be considered malware. Attackers can lurk for a long time without any suspicious behavior. Furthermore, a malicious file could be opened a few days after it was downloaded. Therefore, it is difficult to detect attacks based on contextual information.

To detect such attacks, system event logs should be stored for long term, which requires GBs of hard disk space each day. Several studies [47–49] have aimed to reduce the log size, but such measures can only mitigate this problem because an attack can last for years and data should be stored for a long time. Additionally, it takes time to identify related events by going through these logs. Thus, provenance graphs, also known as dependency graphs or information graphs, are commonly used to traverse logs faster [3, 4, 49]. Real-time detection systems [21, 22] based on provenance graphs usually store them in memory for better performance in terms of computing and graph matching, but the graph keeps growing over time. Because APTs can lie dormant for a long time without suspicious behaviors, these methods suffer from memory problems related to storing the growing graphs and from efficiency problems related to tracking a long-term attack. As shown in Fig. 3(a), it is nearly impossible to perform this type of detection in real-time and for long-term attacks.

In this section, we present a state-based framework. In this framework, each instance of processes and files is analogous to a set of automata, which allows us to detect different attack scenarios in real-time with low overhead by aggregating all the contextual information used to aid detection in every process shown in Fig. 3(b). We first formulate the notions of semantic recognition, data structure, state transition condition and malicious state. Next, we explain the method of reconstructing attacks based our framework.

4.1 Semantic Recognition

Our semantic state definition is inspired by forensic analysis; we automatically recognize the high-level semantics of data flows, control flows and process behaviors. These semantics represent fundamental evidence to be used in context-based detection. We call such semantics atomic suspicious indicators (ASIs).



(a) A provenance graph of an attack.

(b) Aggregate context information to one process in our approach.

Fig. 3. High-level comparison of provenance graph-based approaches with our approach.

An ASI includes one of the following types of semantics, and is detected or inferred by corresponding traces:

1) The *behaviors* an attacker performs to achieve its target. Often detected by *APIs* or inferred by *confidential data flows*.
2) The *source of suspicious codes*, in other words, the reasons why a process can execute such behaviors; inferred by *untrusted data flows*.
3) The ability to conduct *external communications*, inferred by *network activities*.
4) The reasons why a process is executed inferred by *untrusted control flows*.
5) *Additional features* that describe the attack.

Each ASI can be described as a triad: $\langle N_o, T_y, D_e \rangle$. Each ASI is assigned a unique number, N_o, which represents its position in a bitmap to record states. T_y represents the categories, which include the following: 1) suspicious code source, to track potential untrusted code execution; 2) suspicious behaviors; 3) network connections; and 4) features, which are additional features that illustrate different attack scenarios. D_e represents the descriptions used to explain the detection results in human-readable semantics.

These ASIs are recognized by direct extraction from source data or inferred through rules (Sec. 4.3). A new ASI that can aid in the detection of APTs should be declared if 1) it has different semantics than those of existing ASIs or 2) there are multiple ways to recognize the same semantics with different accuracy. Tab. 1 lists a selective set of ASIs. The combinations of different ASIs can ultimately describe different attack scenarios. The following detection steps are based on these ASIs, data flows and control flows.

Some ASIs can be detected easily or generated by inferences based on system event logs. Although the behaviors that are usually performed by attackers are among the most important ASIs, there is no mature method for detecting them.

To address this challenge, we conduct the largest scale study [50] of real-world malware with such ASIs, called remote access trojan (RAT), involving more than 500 white papers [38] and more than 50 RAT families active in the last decade. The result shows that RATs are commonly equipped with tens of ASIs and the implementations of these ASIs are basically the same. We developed methods to detect them on

Fig. 4. A simple example of a screengrab signature. The APIs in the same frame are alternatives that implement similar functions. The signature is generated based on its code implementation, and the sequence is based on the data flows between APIs.

data sources from different layers. To balance the accuracy and efficiency, we manually generated a set of topology API signatures to recognize these behaviors based on their code implementations following the ideas of [51] (e.g., as shown in Fig. 4; the arrows represent the data flows between APIs, and thus, the behavior must be implemented in such a sequence). The coverage and true positive rate of API signatures are high (90%), but they may be ambiguous. For example, the APIs used to take screen shots are also used to draw pictures in windows; even the sequences remain the same (for both use, these APIs are used to copy the content of a device-context from one to another). To address this challenge, we adopt some external information to improve the accuracy (e.g., we find that the API sequence of screen grab can be also used to draw windows, our solution is: when a process is grabbing screens while it has no visible window at that time, we can consider it as a screenshot operation but not a drawing operation). To match these topology API signatures in real-time with high efficiency, we convert these topographies to FSAs to match signatures with streaming data, and these APIs should be matched in a window to reduce the false alarms. In practice, we adopt a time window with 6 seconds. We do not use system calls because they are too low level to reflect the semantics.

API calls are usually recorded by API hooking using sandbox, which cannot be used in real-time detection systems because of its poor performance and stability. We use ETW kernel call stack traces to recover the API calls. ETW can capture a kernel event (including SysCallEnter events, which represent a call to System Calls) along with its call stack. We recover the APIs from call stacks efficiently. However, this process is not our main contribution, we do not describe it in detail here[50].

TABLE 1
A selective list of ASIs. The first column is the ASI number of a process (P) or a file (F). ASIs are categorized by types: code source (CS), behavior (Beh.), feature (Fea.), and network (Net.).

Number	Type	Description
P1	Net.	Network traffic.
P2	Beh.	Access sensitive files.
P3	Beh.	Audio recorder.
P4	Beh.	Keylogger.
P5	Beh.	Execute sensitive commands.
P6	Beh.	Grab screens.
P7	Beh.	Steal Windows credentials from memory.
P8	CS	Injected by untrusted code.
P9	CS	Load unsigned images.
P10	CS	Load unsigned images from the network.
P11	CS	Execute scripts from the network.
P12	CS	Find unknown code in call stacks.
P13	CS	Executed by suspicious threads.
P14	Fea.	No human interactions.
P15	Fea.	Ancestor process has network connections.
P16	Fea.	The process has visible windows.
P17	Fea.	Access data from the network.
F1	-	The file contains data from the network.
F2	-	The binary file does not have a certificate.
F3	-	Contain data from malicious behaviors.
F4	-	The file is sensitive.

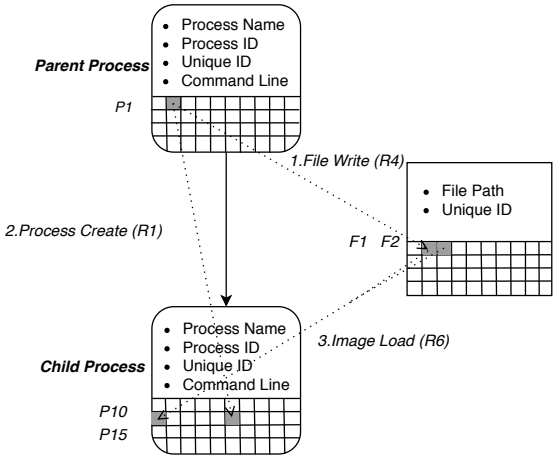


Fig. 5. Data structure and operations. The labels, P1, P10, P15, F1 and F2 are states described in Tab. 1. R1, R2, and R15 are rules described in Tab. 2. This example records the process of a drive-by-download attack.

4.2 Data Structure

To support real-time analysis and long-term monitoring, we propose a main memory FSA-like structure to record the states of each process and file that may be involved in an attack. Please note that we do not need to store any historical events to perform detection, but with the additional goal to reconstruct attacks, we retain only a small set of events that make the states change to a database.

As shown in Fig. 5, we preserve the basic information and states of each process and file in memory when they are in specific states. Each process instance can be described as a quintuple:$\langle N_a, P_i, C_l, U_i, S_t\rangle$, which contains the basic information of a process. N_a is the process name, P_i is the process ID and C_l is the command line. As N_a and P_i can be duplicated, we assign a unique ID (U_i) for each instance. S_t represents the state of this process.

Each file instance is a triad: $\langle N_a, U_i, S_t\rangle$. N_a stands for file path, U_i is the unique ID and S_t is the state. States are predefined, as discussed in Sec. 4.1. Once an instance contains a specific state, the corresponding bit is set as true.

All inactive processes and files are removed from memory to the database to ensure the memory is constant. Files will be recovered only if they are operated by another process.

4.3 State Transition

Our approach is based on the insight that the same types of events have different high-level semantics depending on differences between the subjects and objects involved. For example, reading a downloaded file is different from reading a file existing in a personal directory. The former involves access to an unknown data source and may lead to untrusted code execution, while the latter involves access to personal data and may eventually result in user data leakage. The purpose of this part is to track confidential data flows, untrusted data flows and untrusted control flows. Such analyses are usually performed by forensic analyzers; however, our system performs this analysis automatically.

To automatically distinguish between these events and record the semantics, we created a group of predefined **rules** to assign more detailed semantics to the events. There are a selective set of rules in Tab. 2.

Each rule is a six-tuple: $\langle N_o, S_s, E_v, S_o, D_i, D_e\rangle$. N_o is the serial number of the rule, which is used in an edge to represent how this operation is generated. S_s stands for a specific state of a subject. A subject is always a process in our design. S_o stands for a specific state of an object. An object is a process, a file or an IP. E_v is the event performed by the subject on the object. D_i, forward or backward, indicates the direction in which one entity influences the other. When the subject is in a certain state and performs an event on an object, the state of the object changes in what we call the forward direction, where the subject and object are the source and destination, respectively. By contrast, if the subject is influenced by the object, we call it backward; in this case, the subject is the destination. Note that both S_s and S_o can be one or more states when they are the source. D_e is the description of the purpose of the rule, and it is used to explain the reconstructed attack chain.

Each entity (i.e., subject or object) in our system is like an FSA and can be described as a quintuple:$\langle S, \Sigma, \delta, S_0, F\rangle$.

S**: the set of states.** The combinations of the bits in S_t, which represent the current states of processes and files.

Σ**: the input alphabet.** Consists of the system events E_v.

δ**: the state-transition function.**

$$\delta : S_f \times \Sigma \rightarrow S_l \tag{1}$$

Unlike the traditional original FSA, the state S_f already exists in an entity, and the state S_l is newly generated in another entity involved in this E_v.

S_0**: the initial state.** Once a new process or file appears in our system, we create a corresponding instance in memory. All bits in S_t are set as false. Only files that may contain confidential data are initialized with a state F5.

F**: the set of final states.** Once a process enters one of the states, it will trigger an alert. These malicious states will be discussed in Sec. 4.4.

As shown in Fig. 5, a file is inferred to be downloaded from the network because a process with a network connection writes data to it. Then, a new process loads this

TABLE 2
A selective list of rules. [Beh.] and [CS.] means any ASI of this type.

No.	Subject ASI	Event	Object ASI	Direction	Description
1	P1/P15	start	P15	forward	This process is executed by a process with network connections.
2	[Beh.]	write	F3	forward	A process writes data acquired through malicious behaviors to a file.
3	P17	read	F1	backward	A process accesses data from the network.
4	P1/P17	write	F1	forward	A process writes data from the network to a file.
5	P2	read	F3/F4	backward	A process accesses a sensitive file.
6	P10	load	F1&F2	backward	A process loads a downloaded image with no valid certificate.
7	[CS.]	start	P13	forward	One process is executed by a suspicious process.

downloaded file, and as a result, the process changes into a state that represents this semantic. In other words, the states include the semantics from both the previous object state and this event. Consequently, in our system, we do not need to store any historical events and can detect attacks efficiently.

However, to reconstruct attacks for better understanding and further analysis, we store the events that cause state changes to a database. An event is stored with four attributes: rule number, time stamp, source and destination. The rule number represents the reason why states are changed. The event is like an edge in provenance graphs, but the source and destination of the operation are the bits in the bitmap to record states, not a process or a file. We will discuss this design in Sec. 4.5.

The results in Sec. 5.8 show that existing rules have covered more than 95% of the original events, and less 1% of them are stored in a database for attack reconstruction.

Note that, although we can obtain the offset and length of reading and writing a file, we regard the file as a whole. This simplification can significantly reduce overhead.

4.4 Malicious States

Malicious states are various combinations of individual states that indicate the contextual information needed for detection. As discussed in Sec. 3.2 and Sec. 4.1, ASIs can be categorized into 4 different types: suspicious code sources, network connections, suspicious behaviors and features. If a process contains at least one ASI from each of the first three categories (excluding features), we say it *enters a malicious state*. Each malicious state illustrates a different attack scenario. For example, if a process loaded an unsigned image downloaded from the network, executed malicious behavior and connected to the network, we recognize it as a "download & execution" attack, and we know the attacker's target based on the malicious behavior it executed. If the unsigned image already exists in the host, it can be recognized as "existing malware". Therefore, we do not need to assume that all phases of an attack happen after CONAN started monitoring the system, which we believe is one improvement of CONAN over existing works.

Once a process enters one of the malicious states, our system will raise an alert. In other words, all detection progresses can be performed by simply checking the state of one process. For example, such a check can reveal whether a process is executing unsigned codes from the network. We do not need to know the exact source of the code because that information provides little assistance for detection. In addition, because our system checks only process states, it has much lower overhead than graph-based detection mechanisms [22] but achieves almost the same effects.

We also utilize multiple general features to identify different attack scenarios. For example, the "human interaction" feature reflects whether a process is run automatically, and the "no visible windows" feature indicates whether the user could visibly recognize the existence of this process. If there are more features, there is a higher confidence score of malicious behavior, which is representative of different attack scenarios. The use of more features will help system administrators analyze attacks and reduce false positives. Note that we do not use any white lists of files, processes or domains, except code certification. The malware installed before the deployment of our system can also be detected as a special attack scenario.

4.5 Attack Reconstruction

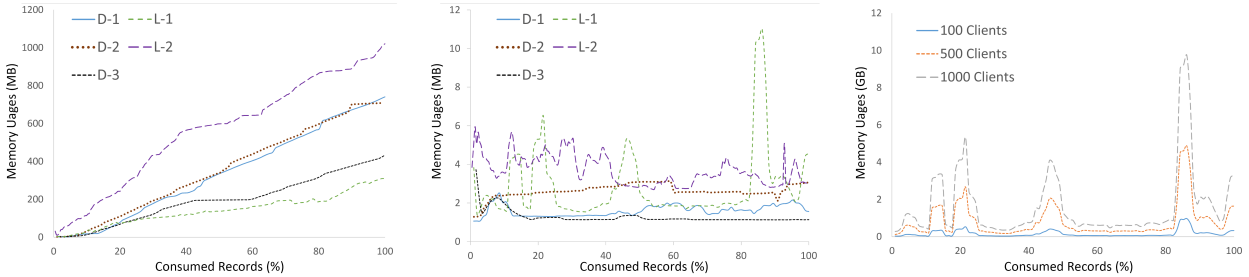
Provenance analysis greatly aids the understanding and detection of attacks. Consequently, we not only detect these malicious attacks but also try to reconstruct these attacks with semantics, similar to the way provenance analysis functions. Such reconstructions help considerably benefit attack analysis, further reducing false positives and helping protect hosts from future attacks.

These tasks can be performed efficiently due to the specificity of our data structure. Because we aggregate all evidence from the target process as states, the basic idea in reconstructing the attack is to explain why the process is classified as malicious, specifically, by backtracking the provenance of ASIs in the process. Because our system preserves the source of every ASI in a database as edges between states (see Sec. 4.2), finding the provenance of the attack can be accomplished in linear time by back tracking through edges. For forward analysis, because our system can detect a malicious process immediately, nearly no additional impact can be inflicted by the suspicious process; thus, there is no dependency explosion for forward tracking.

In practice, a reconstructed graph with dependency explosion contributes less to further analysis. To obtain the graph, we reconstruct only enough evidences to prove that it is an attack instead of trying to reconstruct the whole attack, as tracking attacks with coarse-grained data flows is an unsolved research problem that has been studied for years [4, 5].

5 EVALUATION

We evaluate CONAN on Windows using three different environments with long time running. The results show that CONAN can detect multiple types of attacks with high accuracy and low overhead.



(a) Provenance graphs with a single host. (b) CONAN with a single host. (c) CONAN with different numbers of hosts (L-1).

Fig. 6. Memory footprint vs. % of records consumed. Provenance graph (left) vs CONAN (middle) and CONAN with different numbers of hosts (right).

TABLE 3
Dataset for each campaign with the duration and distribution of different events and the total number of events and call stacks. DARPA (D) and Laboratory (L).

Dataset	Duration (hh-mm-ss)	File Read (Original)	File Read (Optimization)	File Write	File Create /Delete/Rename	Process /Thread	Loadlib	Network	Other	Total # of Events	Call Stack
D-1	8:50:25	45.63%	0.63%	29.60%	15.10%	0.91%	0.64%	7.99%	0.14%	10.66M	107M
D-2	8:48:10	43.33%	0.60%	27.43%	17.00%	0.93%	0.63%	10.65%	0.02%	9.96M	136M
D-3	6:53:40	46.92%	0.38%	25.18%	18.20%	0.87%	0.54%	8.27%	0.02%	6.13M	175M
L-1	3:27:48	20.00%	0.56%	21.62%	24.27%	0.61%	2.65%	30.6%	0.17%	102.5M	201M
L-2	284:36:05	34.40%	0.18%	2.41%	1.73%	0.51%	2.15%	58.75%	0.04%	100M	10B

5.1 Implementation

We adopt Event Tracing for Windows (ETW) as the primary data provider. ETW is a built-in audit system on Windows, and it can provide more than 1000 types of audit logs, including system calls, call stacks, and application-level logs. We extend ETW by adding data sources, such as certificates of binary files, human interactions, and clipboards. The data collector is implemented in C++ and consists of approximately 9.2-thousand lines of code (KLoC). The remaining components are implemented in Java and consist of approximately 6.2 KLoC. All states, rules and malicious states are customizable and specified in configuration files of approximately 112 lines.

5.2 Datasets

We examine our system in three environments: our research lab, DARPA Engagement and with several real-world enterprises. Tab. 3 summarizes the dataset used in the first two environments, and we will discuss the large-scale evaluation in the real-world separately.

The first 3 rows correspond to the attack campaigns carried out by a red team as part of the DARPA Transparent Computing (TC) program. This set spans a period of 23 hours and includes approximately 30 M events and 400 M call stacks. The second two rows of the table correspond to the attack and the benign data collected in our research laboratory, respectively.

These data were collected by our collectors on Windows. The "duration" column in Tab. 3 refers to the length of time that the collector was running on the target machine. Note that the duration covers both benign activities and attack-related activities on a host. The next several columns provide a breakdown of events into different types of operations. File read and write include not only file reads/writes but also some additional data flows, such as clipboards. The number of original file read events is nearly 40% in each dataset, but with the following optimization, the number decreases approximately 100-fold: when a file is read by a process, the collector ignores duplicate read events between two adjacent write events because the states of this file cannot be changed until it is written, and thus, the process will not be influenced by these read events. The process/thread column includes process and thread starts/ends events. The network includes only TCP and UDP packages because ETW cannot provide low-level network events. The "Others" column includes features we collect to better understand attacks, including image certificate, visible windows and human interactions. The number of network events in L-2 is significant because the user watched many movies during testing.

5.3 Environments and Experimental Setups

Our experiments contain three scenarios.

Laboratory Setup. We deployed CONAN on two hosts in our laboratory and implemented the attack described in Fig. 3(a) on one host (L-1), in addition to downloading and installing a set of benign applications with similar behaviors, including IMs (e.g., Skype), remote-access tools (e.g., TeamViewer), image editors (e.g., Photoshop) and others (e.g., PuTTY). Meanwhile, we run our system for weeks on another host to ensure its long-term stability and low false-positive rates (L-2).

DARPA Engagement Setup. The attack scenarios in our evaluation are configured as follows. Three individual hosts installed Windows 10. Because the setup involved an adversarial engagement, we had no prior knowledge about the attack prepared by the red team: we did not know when the attack would occur or what the attack target would be. It is worth noting that, while the red team was attacking the target host, benign background activities were also being carried out on the hosts. These activities

included browsing websites, downloading binary files and executing them, reading and writing emails and documents. In general, nearly 99.9% of the events were related to benign activities. Therefore, the task was to automatically identify attacks amid a large number of benign events in real-time.

In this engagement, we detected all attacks with no false positives. Because in these scenarios, the test team tried to simulate a strict business or government environment, those behaviors that may generate false positives, e.g., unsigned application installation, were not implemented. The reconstruction results do not cover all the activities generated by the attacks. We present part of our results compared to the ground truth data released by the red team in the next section.

Real-world Enterprise Setup We were allowed to deploy our system in three real-world companies for long-term running to verify the stability and efficiency of CONAN. All deployed machines are in their office network instead of their business network for security. All machines can access the Internet and they are operated by employees during office time. The network and machines are also protected by other security productions, such as anti-virus software and firewalls. Accuracy, stability and efficiency can all be examined in such scenarios.

5.4 Attack Scenarios and Reconstruction

In this section, we introduce the detailed detection result of one campaign to illustrate how CONAN works to detect these attacks. The other results will be discussed in Appendix. All detection results with specific states are listed in Tab. 4.

As shown in Fig. 7(a), the attacker redirects the website to another IP address.

1) When the user tries to browse this site, the firefox.exe process will navigate to the fake IP address 138.113.2.43. At this time, *firefox* is labeled P1, which means this process has outside network traffic.
2) Then, the attacker can successfully execute the code remotely using Firefox's vulnerability. When it executes unknown code, CONAN detects it with dynamic call stacks, separates this thread from trusted ones and labels it P12.
3) After *firefox* is compromised, the attacker first executes *hostname* and *tasklist* and reads *Deafult.rdp* to collect information about the host. CONAN records the separated entity *executes sensitive commands* and *accesses sensitive files*.
4) At this time, an alert is triggered as this entity enters a malicious state. This entity includes the states listed in Tab. 4 (D-1 firefox). The feature of having visible windows and interacting with users means that it should be perceived by users. As it has both network connections and executes unknown code, it can be inferred that the executed unknown code may be downloaded from the network. Therefore, this attack is recognized as **"an exploitation of a benign application"**.
5) Then, a binary file, cloud.exe, is downloaded by Firefox. This file is labeled F1, which means it is download from the network.
6) Firefox creates a process, *cloud*, and it loads the unsigned image, cloud.exe. It makes this process *executed by a suspicious process* and *executes untrusted code from the network.*
7) Finally, the suspicious process takes screenshots, executes *whoami*, and sends the screenshots to a new IP address, 78.184.214.212. This attack is recognized as **"download and execute malware"**, which is one of the most common attack scenarios in real-world situations.

TABLE 4
Malicious processes with ASIs.

DataSet	Process	Code Source	Behavior	Network	Feature
D-1	cloud	10,13	16	1	-
	firefox	12	2,5	1	14,16
D-2	dll_loader_x64	10,12,13	7	1	15
D-3	telnet	13	2	1	-
L-1	notepad	8	2,5	1	-
E-1	firefox	9	6	1	14,16

The reconstructed attack chains are shown in 7(b) and 7(c). CONAN automatically explains why these processes are malicious by tracking the provenance of each ASI. The reconstructed attack chains are similar to the original ones.

5.5 Large-Scale Real-World Deployment

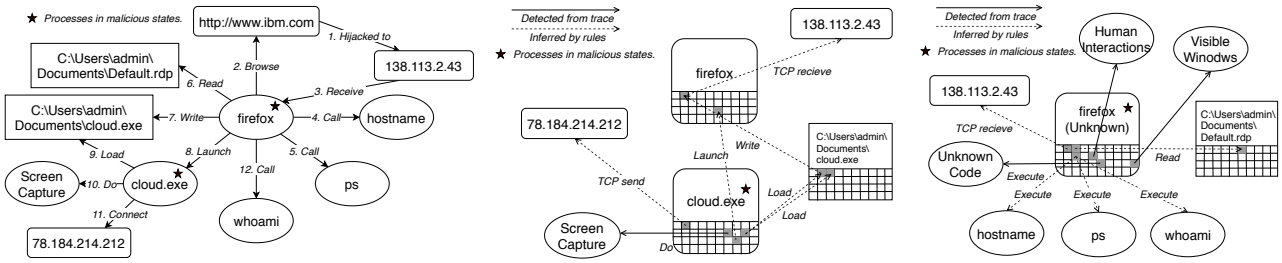
To study the efficiency and stability of CONAN, we deployed it in three real-world companies: a financial company, a communication company and an energy company. We developed our collector on Windows 7/8/10. All of them are 64-bit. Collectors were deployed on 226 office machines in total. These collectors have been running for more than three months, and the result shows that CONAN can run efficiently and stably in real-world scenarios for a long time as we expected.

Meanwhile, we examined the accuracy of our method. There were 422 alarms among 36 million processes, including 14,524 programs, as shown in Tab. 5. As we claimed in Sec. 4.4, we do not assume that machines are clean before the installation of CONAN, and we suppose existing programs on machines can also be malicious. The results are shown in Tab. 5, in which **C** represents attacks with complete attack chains, and **I** represents attacks with incomplete attack chains, in which no initial access phase is detected. Unfortunately, we check all these alarms by using Threat Intelligence[52] and Sandbox[53]. To date, none of them have been classified as malicious. We will discuss the false alarms in the following section.

5.6 False Alarms

To study CONAN's effectiveness in benign environments, we deployed it on the hosts in our laboratory and in three real-word companies.

In the laboratory: Because our detection system focuses primarily on remote-access and suspicious behaviors, we downloaded, installed and ran a set of benign applications with similar abilities and behaviors. We also selected a set of common applications including system applications and other popular applications. We downloaded each application from the network, installed it with the default configuration, and then executed it manually to trigger additional program behaviors (L-1). In addition, we invited



(a) Provenance graph of Attack D-1 (b) Reconstruction of the attack from cloud.exe (c) Reconstruction of the attack from firefox.

Fig. 7. Attack D-1 with reconstruction. The arrows in (b) and (c) represent how to track the provenance of the attacks, instead of the direction of the events.

TABLE 5
The false alarms in the real-world evaluations. In the column Alarm Type, **C** represents attacks with complete attack chains, and **I** represents attacks with incomplete attack chains, especially no Initial Access phase. **I** means the detected suspicious programs were already there before CONAN was installed, while **C** means CONAN captured the whole attack chain, including the Initial Access phase.

Company	Total Machine	Total Process	Total Program	Alarm Type	Alarms	FPR(Alarm/Process) (10^{-5})	Program	FPR(Program) (10^{-2})
Financial	61	9,014,881	4,664	C	15	0.17	9	0.19
				I	98	1.09	68	1.46
Communication	85	16,512,358	7,152	C	17	0.10	8	0.11
				I	124	0.75	73	1.02
Energy	80	11,251,325	5,215	C	22	0.20	11	0.21
				I	142	1.26	91	1.75
Total	226	36,778,564	14,524	C	54	0.15	22	0.16
				I	364	0.99	172	1.28

a volunteer without any understanding of our system to use another host (L-2) for long-term testing. The results show that CONAN can accurately identify benign applications, as no false-positives occurred.

In the real-world scenarios: We deployed CONAN in three companies from different fields, and it has been running for more than three months. Nearly 80% *Untrusted data flows* come from internal network. Most of the programs (more than 70%) which trigger alarms are customized programs in their fields, and the remain ones are common programs, such as firefox(browser) and PuTTY(SSH client). The worst thing is that there are some cracked programs and re-packed ones. It is hard to determine whether they are malicious or gray. We selected one of the false positives for a detailed analysis (shown in Tab. 4, E-1, firefox).

This application already existed when CONAN was deployed, but it could still be a malware instance. Because Firefox is an open source application, we found that it contains 11 blocks of code that implement screen capture; therefore, it may take screen shots for legitimate reasons. For certification purposes, the binary file of firefox.exe was downloaded before CONAN was installed; thus, we were unable to track the Firefox download source. In truth, CONAN cannot separate this scenario from one in which the attacker creates fake applications with normal functions but containing malicious code. For example, an attacker could insert malicious code into an open-source project, recompile it, and upload the malware to induce people to download it.

Tab. 6 shows some important ASIs generated in each dataset excluding true-positive attacks and false positives. Each of them is suspicious and can be part of an attack; for example, P2 is the number of processes that access some sensitive files. However, with these contextual methods and the proposed detection model, there are no more false positives.

5.7 Runtime Overhead and Memory Usage

Our detection system can be separated into two main parts: the collector and the detector. On the client side, the overhead of the collector is negligible: it can run on a host with Intel i5-7500 CPU (4 cores and 3.40 GHz) with less than 10 MB memory and 5% CPU usage. Specifically, Tab. 8 shows the average efficiency of events parsing and signature matching in the collector. Since 100ns is the minimum time granularity on Windows, the result shows that the collector can handle events and APIs efficiently.The bandwidth usage is approximately 1-10 KB/s depending on the workload on the host.

We measure the detection overhead on servers, and the results are listed in Tab. 7. We set up the experiments on a server with an Intel Xeon Silver 4116 CPU (with 12 cores and 2.1 GHz of speed each) and 256 GB of memory running on Ubuntu 18.04. **Memory:** Each detector has constant memory, approximately 2.1 MB of memory on average. Compared to the growing provenance graphs (100 MB at the beginning and 600 MB at the end of a 5-day dataset [22]), CONAN is the only practical system to monitor hosts for long periods. **Efficiency:** The detector uses a single core for one data stream. On average, CONAN can analyze the data 200 to 1500 times faster than the data are generated, as shown in the "Speed-up" column. In other words, CONAN with one core can process many data streams in real time, if CPU was the only constraint. Fig. 6(c) shows the CPU and memory usage when CONAN monitors different numbers of hosts. We confirmed this result by simulating the simultaneous

TABLE 6
A selective list of ASIs generated in each dataset (excluding attacks).

Data set	P1	P2	P3	P4	P5	P6	P10	P11	P15	P16	F1	F2	F3	F4
D-1	2588	26	16	119	10	164	85	5	207	42	8161	17	2	15
D-2	3874	17	7	78	7	99	122	9	171	17	7411	12	7	9
D-3	22	8	47	9	6	25	40	18	26	6	3697	10	2	6
L-1	177	10	7	14	2	20	5	48	463	43	36723	114	19749	11
L-2	2438	12	10	15	5	126350	573	1	5681	126	145161	53	106350	5

TABLE 7
Memory usage and runtime for detection and reconstruction. The numbers in the Speed-up column show that CONAN can consume the data 200 to 1500 times faster than the data are generated.

Dataset	Duration (hh:mm:ss)	Max Memory (MB)	Avg. Memory (MB)	Runtime (s)	Speed-up
D-1	8:50:25	5.40	1.53	51.1	622
D-2	8:48:10	2.29	1.38	47.0	670
D-3	6:53:40	3.55	1.23	30.1	824
L-1	3:27:48	10.46	2.74	45.2	276
L-2	284:36:5	5.68	3.51	659	1555

TABLE 8
The average efficiency of events parsing and signature matching in the collector. (100ns is the minimum time granularity on Windows.)

Event Type	Process Time Per Event (ns)
Process	661
FileIO	<100
Network	<100
API Matching	<100

transmission of different numbers of data streams (L-1) to CONAN in real-time. The results show that CONAN can monitor hundreds of hosts with linear CPU and memory usage.

5.8 Benefit of Semantic Recognition of Events

Since nearly 99.9% of system events are related to benign activities, it is very important to reduce irrelevant data to maintain the efficiency and accuracy of CONAN. As described, we use predefined rules to recognize the high-level semantics of system events. Tab. 9 shows the number of events among different process steps: original events (O), matched by rules (M) and those that cause the state to change and to be stored in a database (R). The numbers in column M are similar to those in column O, which means CONAN tracks most of the original events. The numbers in column M mean that less than 1% are stored for reconstruction. Duplicated *Read* events are prefiltered.

6 DISCUSSION

In this section, we discuss how an attacker aware of the design of CONAN might try to evade the detection mechanism.

In-memory attacks. The addresses in a call stack are the entrance of the next commands. To avoid the suspicious addresses in a call stack, the malicious code cannot call any APIs that will result in kernel events (such as read/write files, system calls and memory operations). But in our experience, the attacker cannot achieve his goal without these APIs. Another possible way is to hook or insert malicious code to benign images; these attacks will also be logged at the beginning because of memory operations. Moreover, it describes a research problem called memory authentication [54]. Another attack CONAN cannot detect is the ROP attack [31], as discussed in Sec. 2.

System Extension. There is more than one way to implement a suspicious behavior, which means we should develop corresponding signatures for each implementation. For example, existing research [55] introduces three methods of taking screen captures on Windows, and there are still some other methods. However, the total number of implementations is limited based on the operating system itself, and it is far less than the number of attacks. Consequently, it is feasible and worthwhile to monitor additional behaviors and their implementations. In addition, our system mainly depends on tracking the information flows to determine why a code is executed and where the sensitive information goes. It is easy to extend our system by adding more data sources and corresponding rules to cover more information flows.

System Recovery. Because our detection method is based on the states, it is important to recover the states when the system crashes or restarts. As all states are stored into a database, they can be recovered from it once crashed. When the state of an entity is changed, it is synchronized to the database. And when an entity is removed, it is marked as out of data in the database. Therefore, when our system restarts, it will recover the in-memory states from the database, and the system is able to continue its work.

White Listing. Our white-listing mechanism based on code certification works well on DARPA engagement and our laboratory, i.e., zero false alarm. However, we get a few false alarms in real-world scenarios. To further reduce these false alarms, we can combine with existing white-listing mechanism with process and/or IP white-listing mechanisms, which are proved to be effective in the literature [21].

Graph Reconstruction. As CONAN retains only the first events that make the states change, it would miss the later events, which have the same effect on an entity. Thus, the reconstructed graph may not be complete. We argue that this method because the main purpose of our system is to detect attacks accurately and efficiently, and the reconstructed graph is only used to understand why a detection signal is raised. Another choice is to delete *duplicate* events when inserting them into the database. In this case, *duplicate* means the sources and destinations of the events, and the matched rules are all the same. To determine whether one event is duplicate, we must **store** and **search** for it, which take both memory storage and CPU computation. In our approach, we only need to check the states of one entity to decide whether to store this event. It is much more lightweight.

TABLE 9
Reduction of events among different process steps: original events (O), events matched by rules (M) and those cause the state changed and to be stored in a database (R).

Data set	Read			Write			Start			Load		
	O	M	R	O	M	R	O	M	R	O	M	R
D-1	66989	64725	19232	3071K	2925k	13K	1429	224	224	2931	5	5
D-2	60064	54544	2513	2652K	2573K	13K	1263	179	179	1924	12	12
D-3	23196	21677	810	1479K	1468K	4K	728	31	31	1383	7	7
L-1	57821	26413	17452	2210K	2203K	37k	976	541	541	10928	171	138
L-2	560K	317K	48175	7415K	7311K	113K	19750	4807	0	54187	1	1
Average Reduction			**14.2x**		**1007x**			**9.8x**			**332x**	

7 RELATED WORK

In this section, we investigate relevant studies and compare them to our approach to highlight the novelty of our methods.

Host intrusion detection techniques can be classified into three detector types: misuse, anomaly or hybrid. *Misuse detection* [56, 57] mainly relies on known attack patterns; the collected raw data are converted into an established format before being passed to the detection module, and the detection module will make a decision. Unfortunately, misuse detection techniques have difficulty detecting unknown attacks (i.e., zero-day attacks). We can see that the knowledge-based approaches [58, 59] rely on a database of attack signatures that requires regular updates, while machine learning-based approaches [60, 61] often lack the ability to generalize. *Anomaly detection* [62–68] is used to detect unknown attacks. The behavior profiles of benign programs are stored and updated frequently. Any deviation from the profile is flagged as a potential attack. Forrest et al. [62] proposed an anomaly detection system that used fixed-length sequences of system calls to define normal behavior for UNIX processes. Shu et al. [68] presented a formal framework that surveyed host-based anomaly detection and discussed various dynamic and static approaches in detail. The advantage of anomaly-based techniques is that they can detect zero-day attacks. However, these approaches will cause many false positives because misuse detection and anomaly detection cannot consider both false negatives and false positives. Considering the weakness of the above two methods, *hybrid techniques* have been presented. In addition to combining misuse detection and anomaly detection techniques, hybrid techniques involve specific policies. Policy-based approaches have been well-designed and are exemplified by SLEUTH [21] and HOLMES [22]. SLEUTH leverages trustworthiness tags and confidentiality tags to define the code and data. HOLMES constructs customized policies to exploit the semantic meaning from each step in the APT attack chain. The above works rarely discussed about the essential intention for attacking life-cycle, rendering some false alerts or false negatives still exist (we have listed these weakness in Sec. 3).

CONAN is rather different than previous work. We propose a detection model that summarizes the three essential phases that exist in an APT attack. Through the model, we can disclose the whole attack chain and accurately detect potential danger.

Provenance tracking aims to discover the complete attack paths in a complex context. Backtracking is a common solution used in previous work [21, 47] inspired by the pioneering work BackTracker [4]. Thereafter, PriorTracker [69] optimizes the process proposed in [4] and enables a forward-tracking capability for timely attack causality analysis. During forward/backward searching, provenance graphs are built to record system object/subject dependencies. Studies [8, 10, 21, 70–72] utilize system call data to track information flows. To improve precision, fine-grained data are collected by the authors of [73–75], but blindly increasing the amount of data leads to an increase in overhead. SLEUTH [21] innovates by using tags for efficient event storage and analysis, but the proposed polices have inherent limitations. While SLEUTH maintains a whitelist of internal IP addresses (DNS lookup, etc.) that are not tagged with untrusted origins, which needs to be maintained frequently to reduce false positives, the tag-based approach is essentially a graph-based storage method, and it is difficult to deal with the long-time APT attacks. Because SLEUTH takes long to process the dependencies of large amounts of data, it is difficult to guarantee real-time performance. The amount of data is proportional to the memory consumption, and growing data will cause a memory explosion.

Unlike previous work, CONAN makes good use of the concept of provenance graph for real-time detection. CONAN employs a novel state-based framework with constant memory usage and low overhead. Our system is context-sensitive and introduces FSM-like structures to automatically transfer the status. Moreover, we can analyze audit data, perform state transition and generate alarms in real time regardless of how long the APT attack will last.

8 CONCLUSION

In this paper, we present CONAN, which provides efficient and accurate APT attack detection using an FSA-like state transition approach. We identify the three most essential components of APT attacks to reduce false positives and false negatives, along with some designs to better recognize semantics. Unlike related studies, we utilize states instead of a provenance graph to record semantics, which ensures constant memory usage over time. During evaluation, with a well-designed detection model, CONAN detected all attacks rapidly with only one false alarm. With the help of our state-based framework, CONAN maintains constant memory usage (1-10 MB) over time, unlike previous methods built on growing provenance graphs.

REFERENCES

[1] "2019 Fireeye Annual Report." https://bit.ly/2Ji320M.
[2] "Cuckoo sandbox." www.cuckoosandbox.org.

[3] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality." in *NDSS*, 2005.

[4] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

[5] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 401–410.

[6] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 101–120.

[7] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The taser intrusion recovery system," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 163–176.

[8] A. Goel, W.-C. Feng, D. Maier, and J. Walpole, "Forensix: A robust, high-performance reconstruction system," in *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*. IEEE, 2005, pp. 155–162.

[9] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *International Provenance and Annotation Workshop*. Springer, 2006, pp. 171–183.

[10] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel." in *USENIX Security Symposium*, 2015, pp. 319–334.

[11] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 259–268.

[12] K. P. Dyer, S. E. Coull, and T. Shrimpton, "Marionette: A programmable network traffic obfuscation system," in *24th USENIX Security Symposium*, 2015, pp. 367–382.

[13] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "Exposure: Finding malicious domains using passive dns analysis." in *Ndss*, 2011, pp. 1–17.

[14] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2001, pp. 130–143.

[15] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2001, pp. 156–168.

[16] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.

[17] A. Lanzi, M. I. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior." in *NDSS*, 2009, pp. 255–264.

[18] Z. Qu, G. Guo, Z. Shao, V. Rastogi, Y. Chen, H. Chen, and W. Hong, "Appshield: Enabling multi-entity access control cross platforms for mobile app management," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2016, pp. 3–23.

[19] N. Ben-Asher and C. Gonzalez, "Effects of cyber security knowledge on attack detection," *Computers in Human Behavior*, vol. 48, pp. 51–61, 2015.

[20] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaing heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1035–1044.

[21] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *Proc. USENIX Secur.*, 2017, pp. 487–504.

[22] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[23] "MITRE ATT&CK," https://attack.mitre.org/.

[24] "M-Trends Reports." https://bit.ly/2q9ItbI.

[25] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[26] M. N. Hossain, J. Wang, O. Weisse, R. Sekar, D. Genkin, B. He, S. D. Stoller, G. Fang, F. Piessens, E. Downing *et al.*, "Dependence-preserving data compaction for scalable forensic analysis," in *27th USENIX Security Symposium*, 2018, pp. 1723–1740.

[27] X. Han, T. Pasquier, T. Ranjan, M. Goldstein, and M. Seltzer, "Frappuccino: Fault-detection through runtime analysis of provenance," in *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[28] X. Han, T. Pasquier, and M. Seltzer, "Provenance-based intrusion detection: opportunities and challenges," in *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[29] M. Salehi and L. Rashidi, "A survey on anomaly detection in evolving data:[with application to forest fire risk prediction]," *ACM SIGKDD Explorations Newsletter*, vol. 20, no. 1, pp. 13–23, 2018.

[30] "2016-enterprise-phishing-susceptibility-report," https://bit.ly/2SikDrR.

[31] H. Shacham, E. Buchanan, R. Roemer, and S. Savage, "Return-oriented programming: Exploits without code injection," *Black Hat USA Briefings (August 2008)*, 2008.

[32] "Control Flow Enforcement Technology Preview," https://intel.ly/2YUkJIT.

[33] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[34] S. Sinnadurai, Q. Zhao, and W. fai Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

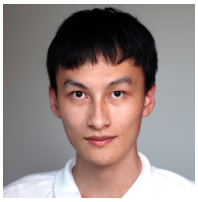
[35] S. S. Vendicator, "A stack smashing technique protection tool for linux," *https://bit.ly/2xRlqXn*, 2000.

[36] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented program-

ming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.
[37] "TACTICS, TECHNIQUES, AND PROCEDURES," https://bit.ly/2Gf5T8u.
[38] "APT Notes," https://github.com/aptnotes/data/.
[39] "APT1," https://bit.ly/2ookhjX/.
[40] "APT38," http://alturl.com/ggn5j.
[41] "Permissions overview," https://bit.ly/2x4HKiW.
[42] "Memory-Based Attacks are on the Rise: How to Stop Them," https://bit.ly/30ytTv0.
[43] "What is a fileless attack." https://bit.ly/2JPszR6.
[44] "Ten process injection techniques," https://bit.ly/2JyGj0d.
[45] "Detecting reflective DLL loading with Windows Defender ATP," https://bit.ly/2Jz4I5Y.
[46] "Process injection in MITRE ATT&CK," https://bit.ly/2Sp3cWU.
[47] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *NDSS*, 2016.
[48] K. H. Lee, X. Zhang, and D. Xu, "Loggc: garbage collecting audit log," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1005–1016.
[49] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 504–516.
[50] H. X. Y. C. Y. C. C. X. Runqing Yang, Xutong Chen, "Ratscope: Recording and reconstructing missing rat semantic behaviors for forensic analysis on windows," in *https://dwz.cn/iiU22qWn*.
[51] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host." in *USENIX security symposium*, vol. 4, no. 1, 2009, pp. 351–366.
[52] "VirusTotal," virustotal.com.
[53] "Hybrid-Analysis," hybrid-analysis.com.
[54] S. Vig, G. Jiang, and S.-K. Lam, "Dynamic skewed tree for fast memory integrity verification," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 642–647.
[55] "Various methods for capturing the screen," https://bit.ly/2JHosmJ.
[56] S. Kumar and E. H. Spafford, "A pattern matching model for misuse intrusion detection," 1994.
[57] O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz, "An intelligent intrusion detection system (ids) for anomaly and misuse detection in computer networks," *Expert systems with Applications*, vol. 29, no. 4, pp. 713–722, 2005.
[58] A.-S. K. Pathan, *The state of the art in intrusion prevention and detection*. Auerbach Publications, 2014.
[59] J. Yu, Y. R. Reddy, S. Selliah, S. Reddy, V. Bharadwaj, and S. Kankanahalli, "Trinetr: An architecture for collaborative intrusion detection and knowledge-based alert evaluation," *Advanced Engineering Informatics*, vol. 19, no. 2, pp. 93–101, 2005.
[60] J. Cannady, "Artificial neural networks for misuse detection," in *National information systems security conference*, vol. 26. Baltimore, 1998.
[61] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*. IEEE, 2005, pp. 118–125.
[62] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 1996, pp. 120–128.
[63] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*. IEEE, 1999, pp. 120–132.
[64] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *sp*. IEEE, 2001, p. 0144.
[65] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 318–329.
[66] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 251–261.
[67] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 305–316.
[68] X. Shu, D. D. Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 4, p. 12, 2017.
[69] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security." in *NDSS*, 2018.
[70] J. Dai, X. Sun, and P. Liu, "Patrol: Revealing zero-day attack paths through network-wide system object dependencies," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 536–555.
[71] X. Xiong, X. Jia, and P. Liu, "Shelf: Preserving business continuity and availability in an intrusion recovery system," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 484–493.
[72] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.
[73] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." in *NDSS*, 2013.
[74] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "Mpi: Multiple perspective attack investigation with semantics aware execution partitioning," in *USENIX Security*, 2017.
[75] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

Chunlin Xiong received his B.Eng. on computer science from Xian Jiaotong University, Xian, China, in 2015. He is currently a candidate of Ph.D. with the college of cyber security, Zhejiang University, China. His research interests include system security, software security and forensic analysis.

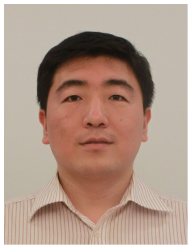
Tiantian Zhu received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China, in 2019. He is currently a lecturer with the college of computer science and technology, Zhejiang University of Technology, China. His research interests include mobile security, system security and artificial intelligence.



Weihao Dong received his B.Eng from Zhejiang University, Hangzhou, China, in 2019. He is currently a master student at UC Berkeley. His research interests include system security and intrusion detection.

Linqi Ruan is a graduate student of Zhejiang university. She mainly interested in endpoint security and reverse engineering.

Runqing Yang received his B.Eng. degree in software engineering from HeFei University of Technology, China, in 2015. He is currently pursuing the Ph.D. degree with the college of computer science and technology, Zhejiang University, Hangzhou, China. His research interests include intrusion detection and attack investigation.



Yueqiang Cheng Experienced Researcher with a demonstrated history of working in the research industry. Skilled in Xen, SGX, Linux Kernel and data privacy. Strong research professional with a PhD focused on System Security, software security, data privacy and hardware security.

Yan Chen received the Ph.D. degree in computer science from the University of California, Berkeley, CA, USA, in 2003. He is a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. Based on Google Scholar, his papers have been cited over 7000 times and his h-index is 34. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. Prof. Chen won the Department of Energy (DoE) Early CAREER Award in 2005, the Department of Defense (DoD) Young Investigator Award in 2007, and the Best Paper nomination in ACM SIGCOMM 2010.

Shuai Cheng received his B.Eng. from Zhejiang University, Hangzhou, China, in 2018. He is currently a graduate student of Zhejiang University, China. His research interests include web security, system security and reverse engineering.

Xutong Chen received his B. Eng degree from Shanghai Jiao Tong University, Shanghai, China, in 2016. He is currently a graduate student of Northwestern University, IL, USA. His reseach interests include system security, blockchain security and forensic analysis.

APPENDIX

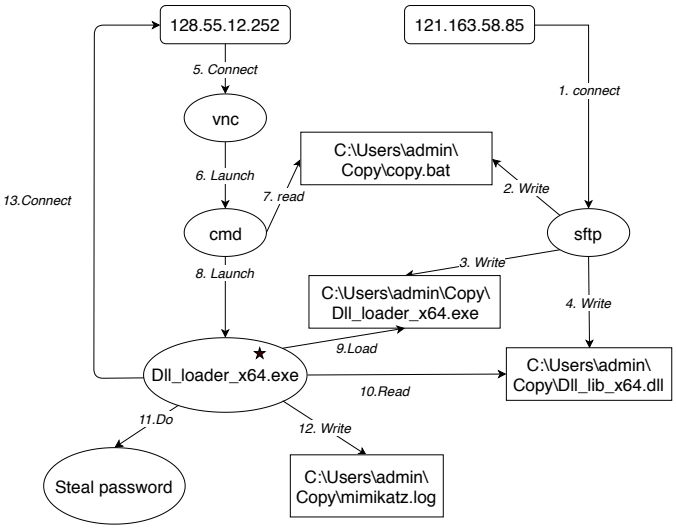
The attack in D-2 is shown in Fig. 8.

1) SFTP connects an external IP, and downloads three files. All these three files are labeled F1. Thus, the detector has known that these files contain data from the network.
2) Then the attacker accesses this machine remotely via VNC server from another IP. In this step, the detector only knows that the VNC has external network connections.
3) The attacker launches a command line to execute the download script. At this time, the detector finds that cmd is a script host, and it reads a file which contains data from the network. So the process cmd is labeled with P11, and the execution is suspicious.
4) A new process is created by suspicious cmd, and loads a downloaded and unsigned image file, Dll_loader_x64.exe. This process is created via a untrusted control flow (P13), and the source of its code is also untrusted (P10).
5) The attacker hides the malicious code in a Dll_lib_x64.dll. And this process executes the malicious code in this downloaded and unsigned image file by *reading* it, instead of *loading* it, implemented by Reflective Loading [45]. But CONAN concentrates on the code executed by this process, and finds that there are unknown addresses in call stacks (P12). Thus, this process is really suspicious.
6) Finally, the process steals Windows credentials from memory (P7), saves the stolen data into logs (F3), and sends them back to the attacker. At this time, this process is created by untrusted thread (P13, code source), executes untrusted code (P10, P12, code source), steal Windows credentials (P7, behavior) and has network connections (P1, network). Thus, it enters a malicious state and will trigger an alert.

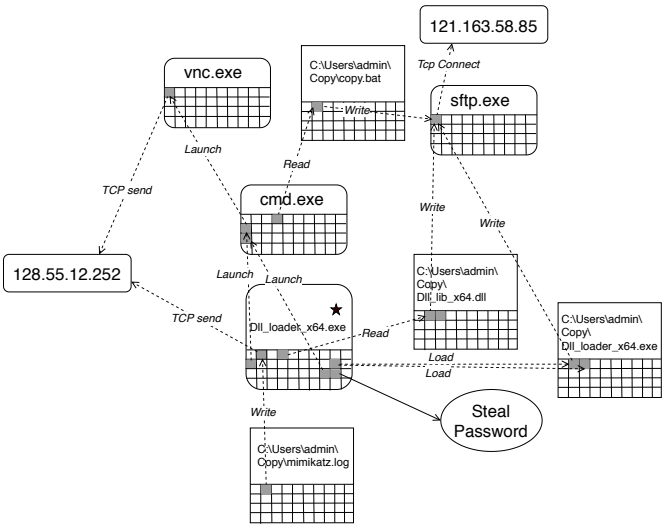
The attack in D-3 is shown in Fig. 9

1) The user downloads a Microsoft Office document with macro from ibm.com (F1).
2) The file is opened by WinWord, and the macro contained in it is executed (P12).
3) A PowerShell process is created by WinWord (P13). It executes keylogger (P4) and save the stolen data into a file (F3). Because PowerShell does not have any network connections, it does not enter a malicious state.
4) PowerShell creates a new process Telnet (P13). It reads the stolen data from the file (P2), and sends it back to the attacker. At this time, this Telnet process is created by untrusted tread (P13, code source), reads sensitive data from file (P2, behavior), and has network connections (P1, network). Thus, it enters a malicious state and will trigger an alert.

The attack in L-1 is shown in Fig. 10. The attack is similar to the one in D-3, while this Powershell downloads a malware instead of doing the malicious behaviors by itself. We cannot automatically reconstruct the whole attack in this scenario as shown in Fig. 10(b). Because we are trying to avoid dependency explosion, and the reconstructed part is already a *complete* attack.

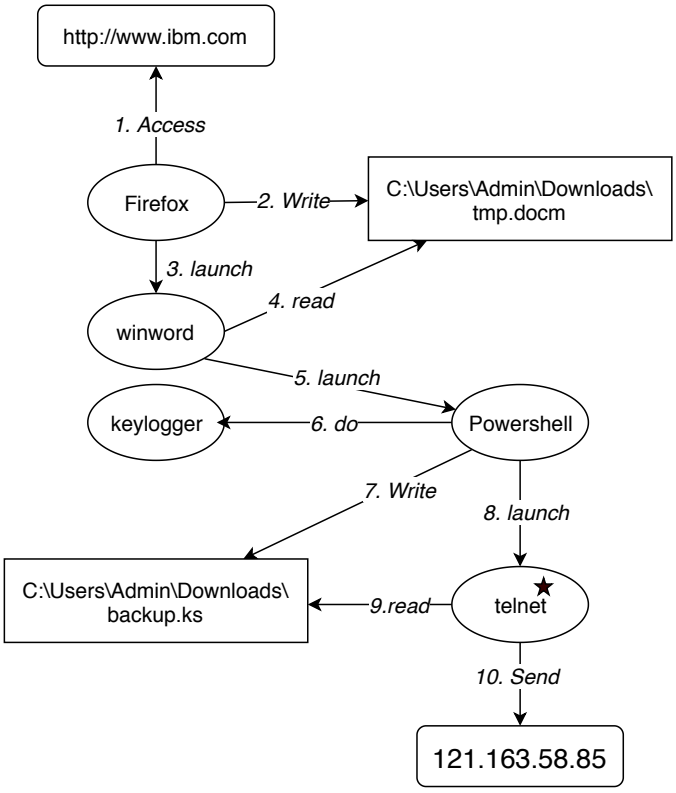


(a) Attack scenario in D-2.

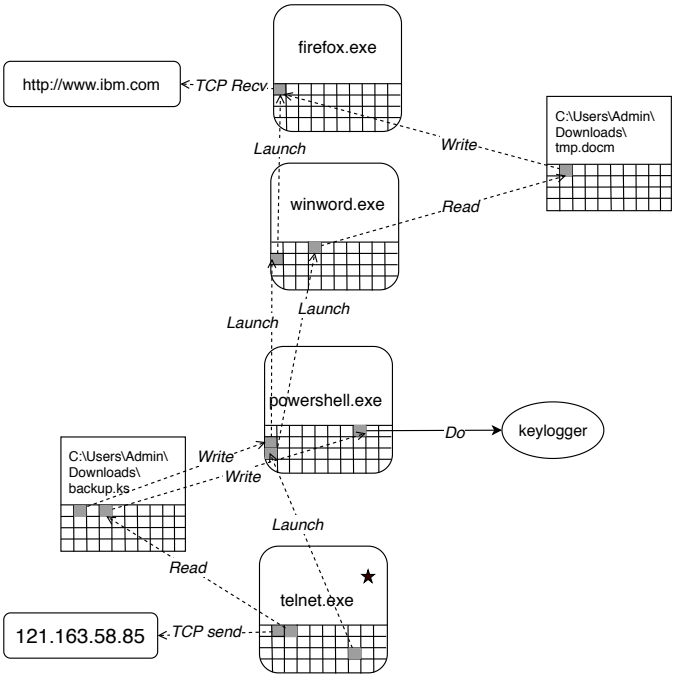


(b) Reconstructed attack in D-2.

Fig. 8. Attack scenario vs. reconstructed attack of D-2.

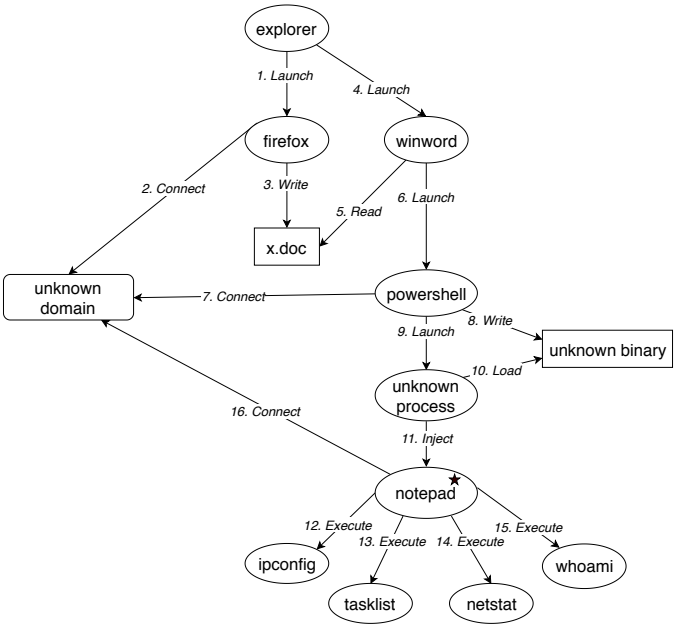


(a) Attack scenario in D-3.

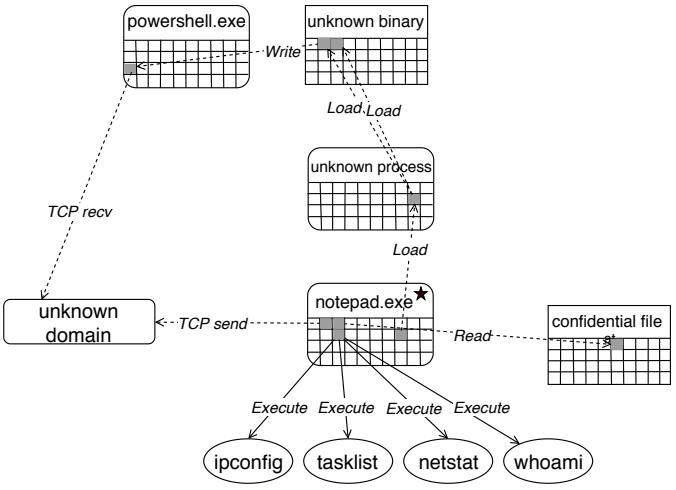


(b) Reconstructed attack in D-3.

Fig. 9. Attack scenario vs. reconstructed attack in D-3.



(a) Attack scenario in L-1.



(b) Reconstructed attack in L-1.

Fig. 10. Attack scenario vs. reconstructed attack in L-1.