Cost-effective Attack Forensics by Recording and Correlating File System Changes

Le Yu, Yapeng Ye, Zhuo Zhang, and Xiangyu Zhang, *Purdue University*

https://www.usenix.org/conference/usenixsecurity24/presentation/yu-le

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Cost-effective Attack Forensics by Recording and Correlating File System Changes

Le Yu
Purdue University

Yapeng Ye
Purdue University

Zhuo Zhang
Purdue University

Xiangyu Zhang
Purdue University

Abstract

Attack forensics is particularly challenging for systems with restrictive resource constraints, such as IoT systems, because most existing methods entail logging high frequency events in the temporal dimension, which is costly. We propose a novel and cost-effective forensics technique that records information in the spatial dimension. It takes regular file-system snapshots that only record deltas between two timestamps. It infers causality by analyzing and correlating file changes (e.g., through methods similar to information retrieval). We show that in practice the resulting provenance graphs are as informative as the traditional attack provenance graphs based on temporal event logging. In the context of IoT attacks, they are better than those by existing techniques. In addition, our runtime and space overheads are only 8.08% and 5.13% of those for the state-of-the-arts, respectively.

1 Introduction

The Internet of Things (IoT) is becoming an integral part of daily life for many people. Smart devices are widely used to provide autonomous control of appliances and convenient human-computer interfaces for various tasks such as online shopping and entertainment, remote health and safety monitoring [23,46,64,65]. These devices are usually connected via networks and hence vulnerable to cyber-attacks [22,24,30,31]. Many of these attacks are very sophisticated and carried out by Advanced Persistent Threat (APT) groups. For example, the VPNFilter incident [51] was a notorious attack in 2018 targeting IoT devices. The US Department of Justice has linked the incident to APT28. The incident affected and damaged over 500,000 devices in at least 54 countries and regions worldwide. VPNFilter operated in multiple stages that included the initial infection using backdoor accounts, the command-and-control communication, and the third stage, in which the payloads (e.g., stealing credentials) were deployed. Similar to other APT attacks on traditional computing systems, the attacks on IoT devices are persistent and stealthy, involving multiple stages and taking a long time (e.g., weeks) to fulfill its malicious objective. Due to the increasing popularity of IoT devices and their roles in many sensitive tasks, ensuring security of these devices is of prominent importance.

An important defense technique is attack forensics, which aims to identify root causes of attacks and assess damages. However, IoT devices usually have limited resources such as storage and data processing ability, posing new challenges for forensic analysis. Specifically, audit logging [20,45,50,58–60,72,73] is a classic technique facilitating attack forensics. It records events such as process creation, file reads and writes, socket sends and receives during system execution and then infers causal relations between these events. During attack investigation, all events that are causally related to some attack symptom event are reported (usually in the form of an *attack provenance graph*) for human inspection. There are a large number of extensions to the basic audit logging, addressing problems such as removing redundant events and causal relations [38,42,52,56,74], reducing logging overhead [57,60], developing query system [32–34,55], graph compression [68,69], and semantic pruning [29,42,63,70]. However, most existing techniques are rooted in audit logging and record information in the *temporal dimension*, where events happen in a high frequency. These techniques hence require substantial storage and processing overhead, well beyond the capacity of IoT devices. For example, simply loading the *cnn.com* front page in *Firefox* incurs millions of audit events.

In this paper, we propose a novel cost-effective forensics technique. It does not record events in the temporal dimension, precluding the major factor in resource consumption. Instead, it records information in the *spatial dimension*, namely, system state information such as file system snapshots. It then infers provenance by correlating states across multiple snapshots. For example, although it does not record file reads and writes, a file copy operation can be inferred by correlating the content of the source and destination files. It does not record process creation events either. However, the execution of a program can be inferred from the presence of process file in the "`/proc`" directory and the behaviors of the exe-

cution can be determined by analyzing both the content of the executable file. To reduce storage consumption, system snapshots are taken in a manner similar to the well-known *copy-on-write* strategy [41]. That is, only deltas are being recorded. The method is particularly devised for the workload pattern of IoT devices. Compared to a traditional computing system, an IoT device often serves for a much smaller number of tasks in a very regular fashion. For example, smart bulbs turn themselves on regularly when the light is dim, and email clients regularly ping the server although file system states only change upon sends/receives. As such, system states often have small deltas over time although the corresponding temporal events may be in a much larger number.

Our contributions are summarized as follows.

- We introduce a new *content provenance graph* (CPG), where nodes denote stateful entities (e.g., files and sockets) and edges denote their causality. We show that CPG is equivalent to the traditional attack provenance graph (APG), in which nodes are processes, files, and sockets and edges denote events in the temporal dimension. Our novelty lies in tracking and analyzing provenance from the spatial dimension whereas most existing works operate at the temporal dimension, requiring logging individual events. We addressed the entailed challenges, e.g., inferring causality without using explicit events, and showed the advantages of this new perspective.
- We propose a cost-effective method to record system state information and infer causality. The inference is based on a novel abstraction of files called *universal file model*, which allows modeling various types of files and their forensics related states. A set of Datalog [25] inference rules is then applied to determine different kinds of causality such as file copy and execution of an application file leading to the creation of another file.
- We develop an algorithm to compute weights for CPG edges and nodes such that a weighted subgraph related to the attack can be determined.
- We develop a prototype ARTISAN (*Cost-effective **A**ttack Fo**r**ensics by Recording and Correla**t**i**n**g File **S**ystem Ch**an**ges*) and evaluate it on five devices (three IoT hubs, a mobile phone, and a personal computer) and compare it to three state-of-the-art attack forensics techniques CLARION [27], ALchemist [74] and eAudit [61]. In the study of 16 attacks collected from the literature, ARTISAN achieves 93.9% precision and 98.8% recall, outperforming the baselines. Our results also show that ARTISAN only incurs 3.68% runtime overhead and consumes 580 MB space for one week, which are only 8.08% and 5.13% of the baselines' overheads, respectively.

Threat Model. We aim to detect attacks that exploit software defects or leverage social engineering techniques to intrude IoT devices and gain certain privileges to deliver their payloads such as exfiltrating confidential data. Similar to many existing attack forensics works [20, 36, 38, 39, 52, 60], we consider hardware attacks (e.g., gaining privileges by exploiting hardware), side channel attacks (e.g., stealing sensitive data by observing CPU power variation) and cryptographic attacks (e.g., recovering private keys in the RSA-2048 schema without extra knowledge) out of the scope of this paper. ARTISAN relies on file system snapshots to gather provenance data. Consequently, attacks that do not leave traces in the file system (e.g., fileless attacks) or that can compromise the file system are not supported by ARTISAN. We consider the Linux kernel, file system, and pre-installed applications can be trusted, and hence file snapshots are part of our trusted computing base (TCB). It is consistent with the assumptions in existing literature [20, 53, 74]. Note that although file snapshots may be manipulated by attackers after they penetrate the kernel, there are a large body of existing works that can be utilized to protect file systems and audit logs [20, 21, 52, 60, 71]. These protection techniques are orthogonal to our work and beyond the scope of this paper. Besides, ARTISAN aims to perform attack provenance for IoT devices or computing systems that have a small budget for logging.

2 Motivation

In this section, we use a real attack example [4] to illustrate how the state-of-the-art forensics techniques work and why they are sub-optimal for IoT attack forensics. We then motivate our design.

2.1 A Real Attack on NextCloud Box

NextCloud Box is a central IoT Hub which connects various sensors in a single house. It collects sensory data and sends commands to devices based on pre-defined action rules (e.g., sending a *"turn-on"* command to smart bulbs when the light sensor detects a low-light situation). A NextCloud Box has its own operation system and applications, interacting with end users and remote devices/hosts through network, just like a regular machine. It regularly connects to cloud to upload and download data. NextCloud Collabora is a popular lightweight online LibreOffice, allowing users to collaborate with others. As part of its core functionality, files of various types (e.g., MP4 and PDF) can be shared among multiple NextCloud Box users. To share data, the user sends a public link of a local office file, e.g., *project.odt*, to other users with the *read-only* permission. However, due to a missing privilege check vulnerability (CVE-2021-32654 [4]) in Collabora, anyone can send a POST request to Collabora to change user permissions. In this case, a remote attacker from *172.16.1.1* crafts a POST request *{type:CHANGE_PERMISSION, ip:172.16.1.1, permission:[read,write,share]}* to add the *write* and *share* permissions for the attacker. Then the attacker inserts a malicious macro to the original LibreOffice file, i.e., *project.odt*. Later when the user opens the compromised *project.odt*, the macro

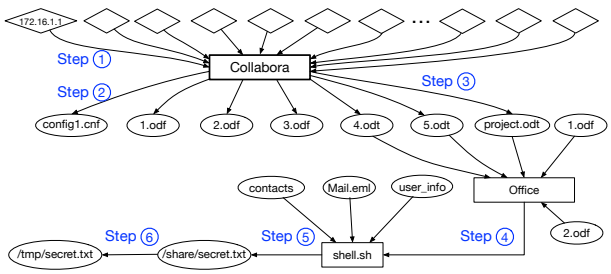


Figure 1: Simplified attack provenance graph generated by ALchemist. Step annotations are manually added for better illustration.

is executed. It sends a notification message to the attacker, drops a malicious script *shell.sh* to the local device, and then executes the script. The malicious script collects information from sensitive files such as *Mail.eml* and *Contacts*. The former contains highly sensitive personal information such as transaction records and insurance policies. The malware also saves such information to */tmp/secret.txt*, which is later copied to */shared/secret.txt* for external access (from the attacker). The attack life cycle may span days and even months. The payload is passive and hence stealthy. Assume after some time, the user observes the presence of the suspicious file and resorts to security analysis to understand the root cause and/or assess the potential damage.

2.2 State-of-the-art Forensics Techniques

Most existing attack forensics methods are based on audit logging which records system level events in the temporal domain. An event describes some operation a subject (i.e., a process) performs on some object (e.g., a file and a socket). These events are very low level, for instance, denoting a process sending and receiving a single packet. As such, events are generated in a very high frequency, demanding substantial resources to store and process them. Linux Audit is the most popular audit logging systems. Since many IoT devices have their operating systems built on Linux, for instance, Android, they can be easily configured to support audit logging. Step ① in Figure 1 denotes an audit event, that is, Collabora handles a request from an external IP *172.16.1.1*. The event yields two nodes, a socket node (a diamond) and a process node (a rectangle) and an edge denoting their causality.

The forensics process is started when some suspicious symptom is observed. The typical procedure entails analyzing raw audit log events, inferring causal relations across events, and building an attack provenance graph by including all events reachable from the symptom event (through causal relations). Specifically, nodes denote processes, sockets, and files. An edge is introduced between two nodes if there is causality. In many cases, an edge corresponds to an audit event. For example, an event that process A reads from file F introduces an edge between A and F. The vanilla Linux Audit and a simple causality inference algorithm introduce a large number of bogus dependences, causing the so-called *dependence explosion problem*. For example, in a long running process like *Firefox*, any file write may be considered causally dependent on all the preceding socket receives as long as they happen within the same process. Such bogus dependences could be at the scale of millions. There are hence various proposals that leverage program instrumentation [52,53,56], high level application logs and UI logs [39,71,74] and statistical analysis [37,38,43,55] to address the problem.

In our attack case, assume the attack symptom is the suspicious file */shared/secret.txt*. Figure 1 shows the attack provenance graph generated by a state-of-the-art technique ALchemist [74]. ALchemist combines built-in application logs and audit logs such that the high-level semantics encoded in application logs (e.g., *tab* opens/closes in *Firefox*'s logs) helps separating low-level system logs to independent *execution units* such that bogus dependences across units can be precluded. However, while mature applications like *Firefox* have well-structured built-in applications logs, applications in IoT devices often do not support informative logs due to the limitation of resource consumption. As such, ALchemist degenerates to a traditional forensics technique that suffers dependence explosion. In the attack provenance graph (Figure 1) as well as the rest of the paper, we use diamonds to represent network nodes, ovals to represent file nodes, and boxes to represent process nodes. Edges follow the direction of data flow. Starting from the symptom file *secret.txt* (on the right), we can back-trace and identify the relevant subjects and objects. Specifically, as the file is generated by *shell.sh*, a process node denoting *shell.sh* is included in the graph, and also all the related objects (e.g., *Mail.eml*). Furthermore, the process *Collabora*, which forks *shell.sh*, is included. However, as *Collabora* reads data from multiple IPs and configuration files, all these IPs and files are included in the graph. Such dependence explosion causes substantial difficulty in locating the root cause IP, i.e., *172.16.1.1*. In addition, it misses the details how *Collabora* writes to *shell.sh*, namely, the remote attacker sends a malicious request, which emits a malicious macro in an *odt* file, leading to the drop of the malicious shell.

Note that instrumentation based techniques that generate additional system events to separate a long running process to execution units (e.g., emitting additional events denoting tab creation/switch for *Firefox*) are hardly applicable in the IoT context because (1) they entail nontrivial overhead; and (2) instrumentation is considered unacceptable in most IoT devices. For example, NextCloud Box has integrity protection policies that prevent any efforts tampering with their applications (e.g., by routine update checks) [10].

To summarize, state-of-the-art techniques can hardly be applied to IoT devices because of (1) the substantial raw log storage requirement and (2) the unique application constraints that disable advanced forensics methods.

2.3 Our Approach

The root cause of the aforementioned limitations is that existing techniques rely on logging high frequency events in the temporal domain. We observe that attack steps are often at a low frequency due to the *low and slow* property [26] of APT attacks. Existing techniques reconstruct attack provenance by computing the transitive closures of causality among high frequency events. We propose a new forensics technique whose basic design is completely different. We propose to take low frequency snapshots in the spatial dimension, e.g., record file system states (at the interval of one minute in this paper). Due to the limited number of tasks an IoT device is responsible for, the deltas across snapshots tend to be small. As such, we only need to record these deltas in order to faithfully recover full system states. The challenge lies in how to precisely infer causality (across attack steps). Inspired by *information retrieval* [62], we propose to reverse engineer causality by correlating state information across snapshots. For example, data/file copies can be recovered by matching file contents. Processes reading/writing files can be disclosed by extracting the file and directory names embedded in the executable sections of the process files. Therefore, we reduce the challenge to a content matching and analysis problem. Since such matching/analysis has inherent uncertainty, just like in information retrieval, we use confidence values and ranking to prioritize important dependences. According to our experiments in Section 4.2, our prototype ARTISAN has 3.68% runtime overhead on the file system benchmark *postmark*, which is only 8% of the overhead of audit logging, and consumes 82 MB storage per day whereas audit logging consumes 1.28 GB.

Figure 2 shows the final provenance graph generated by our system. Our graph has two types of nodes, oval representing files and diamond representing uniform resource identifiers (URIs). It has many types of edges. Specifically, a directed data flow edge denotes information is propagated from a parent node to a child node. It is often annotated with the operation that induces the data flow such as "`ReadFile`" and "`WriteFile`". A directed control edge denotes the access to a child node. It is often annotated by the access type such as read ("`R`") meaning a parent node has read permission on a child node, write ("`W`") meaning a parent node has write permission on a child node, and access control ("`X`") meaning that a parent node specifies the permissions to a child node. An undirected matching edge with annotation "`M`" denotes that two files have partial content match.

The graph clearly captures the attack provenance. At step ①, the control edge from *config1.cnf* to *project.odt* denotes that *config1.cnf* is changed such that the permissions of *project.odt* are changed. It is also denoted by the control edge from *172.16.1.1* to *project.odt* as the former has both read and write permissions on *project.odt*. Note that such changes can be detected by analyzing file differences. At step ②, the

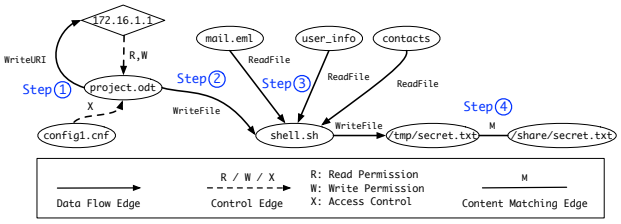


Figure 2: Provenance graph generated by ARTISAN, the graph has 9 nodes and 9 edges with 100% precision and 100% recall. Each file is denoted by an oval and an URI by a diamond.

data flow edge between *project.odt* and *shell.sh* indicates the former writes to the latter. ARTISAN determines this relation by analyzing the executable code sections of *project.odt* and observing a file write system API with *shell.sh* being the destination. At step ③, *shell.sh* reads from multiple other files such as *mail.eml* and *contacts*, and writes to */tmp/secret.txt*. These edges are determined by analyzing the executable *shell.sh*. At step ④, the copy from */tmp/secret.txt* to */share/secret.txt* is disclosed by the content matching edge between the two. The whole provenance graph has 16,276 nodes and 11,072 edges whereas the subgraph has close correlations with the attack symptom, as determined by weight values computed by ARTISAN, has only 9 nodes and 9 edges. It has a precision of 100% and a recall of 100%. In comparison, the attack provenance graph by ALchemist has a precision of 52.9% and a recall of 81.8%. In Section 3.3, we will discuss how to derive the various kinds of edges and determine their happens-before relations.

3 System Design

The workflow of ARTISAN is shown in Figure 3. For a given computation system (e.g., an IoT device or a mobile device), ARTISAN regularly acquires snapshots of file systems generated by the *Zettabyte File System* (ZFS) and determines a list of changed files (step ①). As these files have different types and hence various formats, we use a *universal file model* (UFM) to represent them. The uniformed representation consists of a set of pre-defined (uniform) data types and basic relations, and the parsing/transformation from a specific file type to the UFM is based on a set of pre-defined rules (step ②). During attack investigation, given a symptom, ARTISAN applies a set of pre-defined inference rules to infer the set of files and sockets that have direct/indirect causal relations with the symptom and produce a *content provenance graph* (CPG) (step ③), from which the attack lineage can be disclosed. To accurately identify the nodes highly related to an attack, ARTISAN assigns a weight to each edge in CPG according to file content similarities and their semantic couplings (step ④). Node weights can then be computed starting from the symptom node and based on the edge weights (step ⑤). Nodes

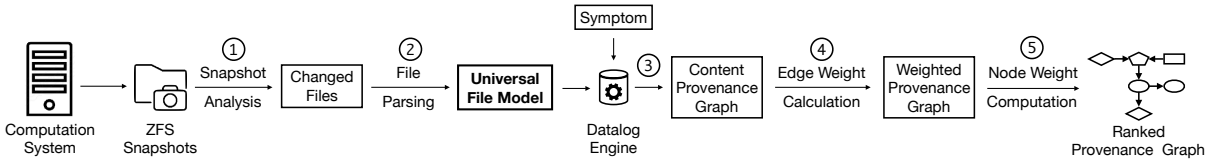


Figure 3: ARTISAN's workflow

with large weights are considered related to the symptom and part of the attack path.

In the following, we first formally define CPG and compare it with the traditional attack provenance graph. We then discuss the individual steps of ARTISAN.

3.1 Content Provenance Graph

Our technique records information in the spatial dimension instead of the temporal dimension. It yields a provenance graph different from the traditional attack provenance graph (APG). Our provenance graph is formally defined in the following.

Definition 3.1 (Content Provenance Graph (CPG))**.** *A CPG is defined as $G = \langle N, E \rangle$, with N the set of nodes and E the set of edges, $N := N_\phi \mid N_\chi$ and $E := E_d \mid E_c \mid E_m$. Here, $N_\phi := \langle t_1, t_2, f \rangle$ denotes a delta of file f between two snapshots at t_1 and t_2, respectively; $N_\chi := \langle t, l \rangle$ denotes a network connection to URI l at timestamp t; $E_d := n_1 \xrightarrow{\tau_d} n_2$ denotes a data flow edge from node n_1 to n_2 and τ_d data flow annotation (e.g., file read/write); $E_c := n_1 \xrightarrow{\tau_c} n_2$ denotes a control edge from node n_1 to n_2 and τ_c the control annotation (e.g., n_1 defines the permissions for n_2 and n_1 invokes a function defined in n_2); $E_m := n_1 \leftrightarrow n_2$ denotes an undirected content matching edge between n_1 and n_2.*

Our graph has two types of nodes and three types of edges. Intuitively, a file node for f is created if f is changed at a timestamp t_2, and the node also records the timestamp of the last delta t_1. The timestamp information is used to determine happens-before relations in causality inference. If f is newly created, $t_1 = 0$ and t_2 is the current timestamp. In Figure 2, the file node *secret.txt* denotes that *secret.txt* has been created. A network connection node with a URI l is created when some file is correlated with a download record involving l (e.g., from the webpage cache file of *Firefox*), an email is received from l, which can be determined from the meta information of the email file, or some configuration file delta indicates that l has been granted access to some local file (e.g., the node *172.16.1.1* in Figure 2). Note that the introduction of network connection nodes is quite different from that in traditional audit logging, in which socket nodes are introduced upon socket creation system calls. Our design is passive and does not require actively monitoring network level events. It introduces connection nodes by observing file changes (e.g., email files, cache files, and configuration files).

A data flow edge is introduced from a process file node n_1 (i.e., a file in directory `/proc` denoting an active application) to a data file node n_2 if some file write operation to n_2 can be statically found in the body of n_1, and n_1's timestamp precedes n_2's. A control edge is introduced from file n_1 to file n_2 if (1) n_1 configures n_2's access control (e.g., the edge from *config1.cnf* to *project.odt* in Figure 2), or (2) n_1 invokes some function defined in n_2, and n_1's timestamp precedes n_2's. A control edge is introduced between a network connection n_1 to a file n_2 if the URI denoted by n_1 is granted permission(s) to n_2 (e.g., the edge from *172.16.1.1* to *project.odt* in Figure 2). An (undirected) content matching edge is introduced between two files if their contents have similarity and their timestamps do not disclose any happens-before relation. Later in this Section, we will discuss how to assign weight to edges to denote uncertainty. For now, we assume all edges are deterministic for discussion simplicity.

Definition 3.2 (Content-Forensics-Ready Application)**.** *We say an application is* content-forensics-ready *if it satisfies the following conditions: (1) its file I/O information such as file name accessed can be precisely extracted by statically analyzing its code body or its auxiliary files; (2) its control transfer to another application or library file can be precisely extracted by statically analyzing its code body or its auxiliary file(s); (3) if its network communication with some remote entity yields any side-effects on the local file system, the URI of the remote entity is visible in some local auxiliary file.*

Here, the concept of application is general. It means a file or part of a file that is executable, including a software app from some app stores, a web app in the form of browser extension, an executable script embedded in some document, and a library. Intuitively, an application is content-forensics-ready if its forensics related behaviors leave trails in the file system and such trails can be extracted by analyzing file system snapshots. An application interacts with other applications through data or control flow. The first condition in the above definition dictates that such data flow is visible to ARTISAN, whereas the second condition ensures that control flow is visible. The third condition specifies that data flow through a network connection is visible. An application may be completely or partially content-forensics-ready. In IoT systems, most of the popular applications are content-forensics-ready. Table 1 shows a list of most popular applications from Nextcloud Box app store [12] and Google Assistant app store [9]. The sec-

Table 1: Content forensics readiness for 16 popular applications used in Nextcloud Box [12] and Google Hub [9]

	Application	Forensic Ready	Readiness for Functionality I/O	URI	CT$^+$	Description
Nextcloud Box	Mail	fully	●	●	✓	send/receive email
			✓	–	–	fetch/update mailbox
			●	●	✓	upload/download attachment
			✓	–	–	add/delete contacts
	Chat	fully	●	●	✓	create/delete conversation
			●	●	✓	send/receive message
			✓	–	–	add/delete participant
	Social	fully	●	●	✓	send/receive message
			●	●	✓	share/delete media
			✓	–	–	create/delete account
	PDF Viewer	fully	●	–	–	open/close document
			–	●	✓	open link
			–	●	✓	submit form
	Passman	partial	✓	–	–	add/delete password
			✗	–	–	encrypt/decrypt password
	Collabora	fully	●	●	✓	share/upload/download doc.
			✓	–	–	add/delete participant
	Video Player	fully	●	–	–	open/close video
			●	●	✓	download video
	Bash	fully	●	–	●	execute command
Google Hub	Firefox	fully	–	●	–	open/close webpages
			●	●	–	download/upload files
			●	●	–	install/uninstall extensions
			●	–	–	add/delete bookmarks
	Thunderbird	fully	●	●	–	send/receive email
			✓	–	–	fetch/update mailbox
			●	●	–	upload/download attachment
	Telegram	fully	●	●	–	create/delete conversation
			●	●	–	send/receive message
			✓	–	–	add/delete contact
	Snapchat	fully	●	●	–	send/receive message
			●	●	–	share/delete media
			✓	–	–	create/delete account
	Okular Document Viewer	fully	●	–	–	open/close document
			–	●	–	open link
			–	●	–	submit form
	KeePass	partial	✓	–	–	add/delete password
			✗	–	–	encrypt/decrypt password
	LibreOffice	fully	●	–	–	open/close document
			–	●	–	open link
	VLC Player	fully	●	–	–	open/close video
			●	●	–	download video

$^+$CT: control transfer, ✓ : information can be extracted by statically analyzing its code body, ● : information can be extracted by statically analyzing its auxiliary file, ✗ : information can not be extracted, – : do not contain such information

```
1  function fetchMessages() {
2    ...
3    filepath = '/usr/share/
4  nextcloud/app/mail/mailbox'
5    ...
6    data = readFile(filepath)
7    return data
8  }
```

(a) OutboxService.js

```
9  function uploadLocalAttachment(url,
10 file) {
11   ...
12   data = new FormData()
13   data.append('attachment', file)
14   ...
15   return post(url, data)
16 }
```

(b) AttachmentService.js

```
17 From: Alice <alice@outlook.com> To: Bob <bob@outlook.com>
18 Date: Mon, 26 Sep 2022 01:44:35
19 ...
20 Content-Type: application/pdf; filename="invoice.pdf"; size=1747375;
21 creation-date="Mon, 26 Sep 2022 01:44:30 GMT";
22 modification-date="Mon, 26 Sep 2022 01:44:35 GMT"
23 ...
```

(c) INBOX.msf

Figure 4: Source code and mail summary file for the NextCloud Box's mail app

ond column shows if they are fully ready. The sixth column shows the forensic-related functionalities. The third, fourth, fifth columns show if each such functionality is File I/O, URI, and control-transfer forensics ready, respectively. Observe that 14 out of 16 applications are fully forensic-ready. The two partially-ready applications utilize encryption and decryption functionalities. As such, the relative auxiliary files are encrypted and the information can not be recovered. Also note that we assume most of these benign applications have plain-text code bodies and auxiliary files. Further discussion is available in Section 5.

Example. As over 10 billion emails are opened daily on mobile/IoT devices [1], the attacker tends to utilize emails to deliver malicious payloads. In this example, we study the forensics-readiness of Nextcloud Box's mail application. Figure 4(a) shows a code snippet from *OutboxService.js*, which specifies how to read/write the local mailbox. In particular, the location of mailbox is specified in line 3. Static analysis can easily derive the set of constant values for the parameter of file I/O API *readFile()* at line 6. Figure 4(b) shows another snippet from *AttachmentService.js*, which specifies how to upload/download attachments. In particular, the parameters of an attachment upload API *post()* depends on user input (e.g., which attachment the user chooses). Hence the file accessed cannot be statically extracted by analyzing the source code. However, it can be extracted from the mail summary file. Figure 4(c) shows the mail summary file *INBOX.msf*. Line 17 shows the target URL *bob@outlook.com* and line 20 the uploaded attachment *invoice.pdf*.

Definition 3.3 (Content-Forensics-Ready File)**.** *We say a (document) file is content-forensics-ready if it satisfies the following conditions: (1) it is not encrypted and (2) its content can be properly parsed.*

Content-forensics-ready data files allow us to correlate their contents. Any encrypted files or files whose formats are not public are not forensics ready. Most popular files on IoT devices are content-forensics-ready. Table 2 shows a list of

Table 2: 25 popular file types for IoT systems

File Type	Metadata	Keyword	Callable Structure	# of APIs modeled
Firefox Extension	✓	✓	✓	64
Chrome Extension	✓	✓	✓	56
HTML	✓	✓	✓	37
Microsoft Mail (MSG)	✓	✓	✓	21
Apple Mail (EMLX)	✓	✓	✓	17
RTF	✓	✓	-	-
XML	✓	✓	-	-
JSP	✓	✓	✓	20
PHP	✓	✓	✓	26
DEB	✓	✓	-	-
PDF	✓	✓	✓	34
Microsoft Excel	✓	✓	✓	40
Microsoft PowerPoint	✓	✓	✓	27
Microsoft Word	✓	✓	✓	42
Office Document	✓	✓	✓	54
Office Presentation	✓	✓	✓	54
Office Slide	✓	✓	✓	54
DSL	✓	✓	-	-
GZIP	✓	✓	-	-
Calendar	✓	✓	-	-
TAR	✓	✓	-	-
Office Spreadsheet	✓	✓	✓	54
Flash	✓	✓	✓	23
RAR	✓	✓	-	-
Shell Script	✓	✓	✓	14

popular document types on IoT devices [22]. All of them can be properly parsed by our system. Furthermore, ARTISAN's architecture is designed to reduce the manual effort required to integrate new file types. A detailed discussion is available in Section 5.

Comparison between CPG and APG. Assume all files have a life span longer than our snapshot interval, meaning that they are not created and then immediately removed within an interval. *If all applications and files are forensics ready, any edge/node in an APG denoting true causality corresponds to some edge/node in the CPG.* In other words, CPG is at least as informative as APG. Specifically, process nodes in APG correspond to process file nodes in CPG. Socket nodes in APG have the corresponding network connection nodes in CPG if the packets received from these sockets are eventually saved to some files (e.g., downloaded files and emails). Some APG socket nodes only lead to behaviors in the memory and thus do not have correspondence in CPG. However, memory read and write operations are invisible to audit logging either [20, 38, 50, 56], and hence attack steps through memory are beyond the scope of APG as well. Similarly, we can infer APG edges denoting real causality have correspondence in CPG. Detailed discussions are elided.

In many cases, CPG has better precision than APG because it has better handling of dependence explosion. Specifically, audit events do not consider contents. For example, packet message bodies or bytes read-from/written-to a file are invisible in audit events. As a result, existing techniques have difficulty determining if some output event (e.g., a file write by *Firefox*) is causally related to a preceding input event (e.g., a socket received by *Firefox*) from audit logs. As such, they often assume the output event is dependent on all preceding input events. In contrast, ARTISAN relies on content analysis and matching, precluding such bogus dependences. Our example in Section 2 illustrates the advantage. In the APG in Figure 1, process Collabora has 86 dependence edges with various sockets such that the write event to *project.odt* is dependent on all the preceding socket receives, including those regular server pings. In contrast, in the CPG in Figure 2, only the edge between the root cause IP *172.16.1.1* and *project.odt* is introduced. Hence a lot of bogus dependence can be pruned. Our results in Section 4.4 also demonstrate the advantage.

A caveat is that attackers may hide their trails by immediately deleting files after payload delivery. As we will discuss later in the section, the file system has built-in support for accessing deleted files.

3.2 Snapshots and File Normalization

In this subsection, we discuss the first two steps of ARTISAN (Figure 3), namely, taking regular snapshots of file system and normalizing file contents for downstream analysis.

Taking File System Snapshots Using ZFS. ARTISAN is built on ZFS [18], which is an advanced next-generation file system designed to support large and safe data storage. It leverages copy-on-write and periodical snapshots to roll back and recover files when a system crashes. According to [11], it is among the top-5 most widely used file systems, and many believe that it will replace *EXT4* in a few years. The reason why ZFS is not integrated into the Linux kernel and has not become popular is due to license compatibility issues [19]. We leverage the built-in ZFS auto-snapshot tool *Sanoid* [15] to take a snapshot every minute, Due to the copy-on-write strategy, a snapshot essentially records only the file system delta within the interval. According to our evaluation in Section 4.2, ARTISAN only incurs a negligible runtime overhead and a small space overhead.

Although ZFS's snapshots contain only deltas, ZFS can internally reconstruct the complete file system state at any timestamp such that users can query. Given two consecutive snapshots, ARTISAN leverages ZFS's command *"ZFS diff"* to identify a set of changed files $\{\langle File, Op\rangle\}$, in which *Op* indicates how the file is changed. The change can be `create`, `modify`, `delete`, or `rename`. For the `create`, `modify`, and `delete` types of changes, ARTISAN further computes the changes, parses and normalizes them. For `rename` changes, ARTISAN unifies the old and file names to a new unique identity, which is used for further analysis. To avoid quick creation and deletion (used by the attacker to remove his trails), we set the *nounlink* attribute in ZFS, which means that when a file is deleted by syscalls like *unlink*, ARTISAN can still access the file.

Parsing and Normalizing Files. ARTISAN supports 25 popular file types as shown in Table 2. Besides, it supports native application files, including Linux ELF executables and Dalvik

File	f	:= (*Metadata* | *Keyword* | *CallableStructure* | *API* | *RawData*)*
Metadata	e	:= ⟨DataSource, Permission, Location, ...⟩
Keyword	k	:= IP | Mail | File | ⋯
CallableStructure	s	:= Button | Sidebar | Context Menu | ⋯
API	m	:= Function
RawData	d	:= (Byte)*

Figure 5: Universal file model

applications (i.e., mobile apps). According to [22], they are the most popular kinds of files on IoT devices. For files that we do not understand their format, we simply treat them as raw bytes. For application files, ARTISAN leverages existing analysis engines Androguard [2], Ghidra [8], and Frida [7] to parse them to their intermediate representations. We further develop program analysis to extract file accesses and external function invocations. These analyses are standard [16, 67] and hence their details are elided. For document files, we mainly build on open-source parsers. The parsed application and document files are further normalized to a *universal file model* that provides a uniform abstraction to simplify downstream analysis, analogous to how *intermediate representation* (IR) simplifies downstream analysis inside a compiler.

Universal File Model. We parse all files to a uniform representation called *universal file model* (UFM), whose definition is in Figure 5. Each file f is parsed to a finite number of records of five different types. The first type of record is ***metadata*** e that holds summary information of a file. It includes the data source (e.g., who created it), access permissions, etc., which are useful in forensics. Specifically, we acquire these information based on a widely used metadata parser *ExifTool* [6]. A ***keyword*** record k contains a value of some pre-defined general type, such as IP and email address. Keywords are recognized using a set of pre-defined regular expressions. An ***API*** record is a standard interface function invocation by a file to carry out some tasks (e.g., requesting some web resource). These records are mostly present in application type of files. Specifically, all function invocations are extracted by the corresponding analysis tools. ARTISAN only focuses on external functions as they can be invoked beyond a file. A ***callable structure*** s denotes a piece of semi-structured data which implies some API invocations. Consecutive bytes that cannot be parsed by ARTISAN are considered a ***raw data*** record. Note that UFM is proposed based on expert domain knowledge and our substantial manual efforts in understanding various file types. Our model is customized for the forensic-related states in ZFS. Better designs may exist, and we will explore these in future work. In Appendix A, we use a PDF example to illustrate the abstract types used in UFM.

3.3 Content Provenance Graph Construction

After parsing and normalizing files, the next step is to construct content provenance graph (CPG) from the normalized UFM records. This entails two sub-steps. The first one is to derive a set of primitive relations called *atoms*. The second is to derive CPG edges by iteratively correlating atoms. The correlation is driven by a set of datalog rules [47].

Atoms

/* Network API */		
(A1)	$readURI(m, URI)$	: API m accesses remote resource *URI*
(A2)	$writeURI(m, URI)$	: Write to a remote *URI*
/* File system API */		
(A3)	$readFile(m, f)$	: API m reads a local file f
(A4)	$writeFile(m, f)$	: API m writes to a local file f
/* Access control */		
(A5)	$filePermit(f_1, f_2)$	: File f_1 configures the access permission for f_2
(A6)	$uriPermit(URI, f)$	: *URI* can access file f
/* Invocation */		
(A7)	$implicitCall(m_1, m_2)$	: invocation of m_1 implies invocation of m_2
(A8)	$binding(s, m)$	: Structure s invokes m under some event
(A9)	$structAccess(f, s)$	: File f gains access to a callable structure s
/* Inclusion */		
(A10)	$contain(f, e)$	: File f contains meta information e
(A11)	$contain(f, k)$	: File f contains keyword k
(A12)	$contain(f, m)$	: File f contains an API invocation m
(A13)	$define(f, m)$	: File f defines a function m
(A14)	$contain(f, s)$	: File f contains a callable structure s
(A15)	$contain(f, d)$	: File f contains raw data d
/* Content matching */		
(A16)	$match(e_1, e_2)$	: Metadata e_1 matches with e_2
(A17)	$match(k_1, k_2)$	: Keyword k_1 matches with k_2
(A18)	$match(d_1, d_2)$	: Raw data segment d_1 matches with d_2
/* Timestamp */		
(A19)	$earliest(e/k/d)$	: Record e, k, or d has the smallest timestamp compared to all its matches

Inference Rules

(R1)	$UndirectedMatch(f_1, f_2)$	:- $match(k_1, k_2)$ & $contain(f_1, k_1)$ & $contain(f_2, k_2)$
(R2)	$DF\_ReadURI(f, l)$	:- $readURI(m, l)$ & $contain(f, m)$
(R3)	$DF\_WriteURI(f, l)$	:- $writeURI(m, l)$ & $contain(f, m)$
(R4)	$DF\_ReadFile(f_1, f_2)$	:- $readFile(m, f_2)$ & $contain(f_1, m)$
(R5)	$DF\_WriteFile(f_1, f_2)$	:- $writeFile(m, f_2)$ & $contain(f_1, m)$
(R6)	$CTRL\_Exec(f_1, f_2)$	:- $contain(f_1, m)$ & $define(f_2, m)$
(R7)	$CTRL\_Exec(f_1, f_2)$	:- $implicitCall(m_1, m_2)$ & $contain(f_1, m_1)$ & $define(f_2, m_2)$
(R8)	$CTRL\_Exec(f_1, f_2)$	:- $binding(s, m)$ & $contain(f_1, s)$ & $define(f_2, m)$
(R9)	$CTRL\_Exec(f_1, f_2)$	:- $structAccess(f_1, s)$ & $contain(f_2, s)$
(R10)	$CTRL\_Access(f_1, f_2)$	:- $filePermit(f_1, f_2)$
(R11)	$CTRL\_Access(l, f)$	:- $uriPermit(l, f)$

Figure 6: Datalog rules to correlate files

Atoms. Atoms are in the form of $p(x_1, x_2, \cdots, x_n)$, with p denoting a predicate (or a relation), and $x_1, \cdots, x_n$ data fields of the UFM types. The set of atoms ARTISAN considers are listed in the top of Figure 6. Atoms (A1) - (A4) represent relations between API invocations and URI/File. Atom (A1) denotes requesting a remote URI through an API invocation (e.g., invocation of *fetch(URI)* in a Firefox extension). Atom (A2) denotes writing data to a remote URI (e.g., invocation of *storeToURL(URI)* in Office files). A3 denotes reading data from a local file (e.g., *Launch(file)* in *PDF*). A4 denotes writing data to a local file (e.g., *openFileWrite(file)* in Office files). A5 and A6 represent the access control relations. A5 denotes file f_1 configures the access permission for file f_2 (e.g., Fire-

fox extension manifest file specifying the access permission for *content.js*). A6 denotes *URI* has the permission to access f. Atoms (A7) - (A9) denote the invocation relations. A7 denotes that the invocation of m_1 implies the invocation of m_2. For example, the presence of $addListener(m_2)$ inside the code body of m_1 indicates that m_2 will be invoked once m_1 is executed. A8 denotes the binding relation. For example, a button is usually bound with an *OnClick()* API. An application file may acquire the handle of a callable structure, through which it may interact with the structure. This is modeled by A9. Atoms (A10) - (A15) denote the inclusion relations. For example, Atom (A10) $contain(f,e)$ states that a meta information record e is contained in file f. Atoms (A16) - (A18) denote the match relation. For example, Atom (A17) means that *Keyword* K_1 matches with *Keyword* K_2. A19 denotes a UFM record has the earliest timestamp (among all the matching records). These atoms can be derived by post-processing the UFM records and simple program analysis.

CPG Construction by Datalog Inference. CPG is constructed by iteratively applying a set of datalog inference rules to derive additional relations from atoms. The derived relations denote edges in CPG. Nodes are also implicitly introduced. Inference rules are in the following format.

$$H :- B_1 \ \& \ B_2 \ \& \ \cdots \ \& \ B_n$$

Specifically, H is the target relation, and B_t is an atom or a relation derived previously. It means that the presence of atoms/relations $B_1, B_2, \cdots, B_n$ leads to the introduction of H. The ultimate goal of these inference rules is to derive relations between two files or between a file and a URI. The set of rules used in ARTISAN are presented in the lower half of Figure 6. Rule R1 introduces an undirected match edge between two files when the two share a common keyword. Rules to match files by matching meta information and raw bytes are similar and hence elided. R2 introduces a directed match edge denoting dataflow between two files when the files have matching keywords k_1 and k_2 and k_1 has the smallest timestamp (i.e., the origin). Intuitively, the copies of a record must come from the original record. Here, the prefix DF_ denotes a dataflow edge. R2 and R3 introduce dataflow edges between a file and a URI, when the file contains a method that accesses the URI. Similarly, R4 and R5 introduce dataflow edges between two files. Besides data flow edges, there may be control edges. Rules R6-R11 denote such edges (with the prefix CTRL_). Specifically, R6-R9 denote that file f_1 causes the execution of f_2. R6 says that f_1 causes the execution of f_2 when f_1 invokes a function defined in f_2. R7 says that the invocation could be implicit (like an invocation of a message send function in f_1 causing the execution of an event handler function defined in f_2). R8 captures the execution relation introduced by a callable structure, e.g., a button and its event handler. R9 denotes that f_1 may acquire the handler of a callable structure s, which indicates f_1 may cause execution of f_2 that contains s. R10 and R11 introduce the access configuration type of control edges. They are directly derived from atoms A5 and A6.

Demand-driven Datalog Inference. ARTISAN relies on the underlying Datalog Souffle inference engine [47] to infer relations between files. However, according to our experiment in Section 4, on average one hundred thousand atoms can be generated everyday with a regular workload. Complex attacks may span days, weeks and even months. It is infeasible for the engine to operate on the atoms from such a long period. We leverage the observation that although an attack life cycle may be long, the attack behaviors may only be relevant to a very small portion of the atoms. Hence the construction of CPG is demand-driven. Particularly, for a backward forensic task that tries to identify the root cause of an attack, ARTISAN starts with the whole set of atoms (for a long period of time) and the symptom file. Only edges (and implicitly nodes) that are directly or transitively correlated with the symptom file are introduced.

3.4 Weighted Content Provenance Graph

Up to this point, our CPG is unweighted. However, a CPG may contain a large number of edges with many irrelevant to the attack. In this section, we discuss how to assign initial weights to edges and propagate these weights to nodes. As such, we could focus on the subgraph with large node weights.

Assigning Importance Values to UFM Records. First, ARTISAN assigns importance values to all UFM records. Inspired by importance value assignment in information retrieval [62], we use TF-IDF for importance of UFM records. Intuitively, the importance of a record r for its enclosing file f is proportional to r's occurrences in f and the uniqueness of r across all files. That is, it measures how representative r is for f. Its formal definition is as follows.

$$TF(r,f) = \frac{f_{r,f}}{\sum_{r' \in f} f_{r',f}}$$

$$IDF(r,F) = \frac{|F|}{|\{f' \in F : r \in f'\}|}$$

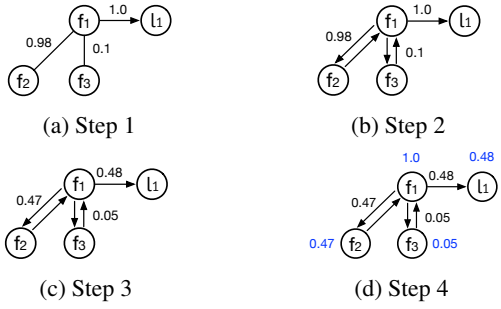
$$TF\text{-}IDF(r,f) = TF(r,f) * IDF(r,F)$$

$TF(r,f)$ denotes the frequency of a UFM record r appearing in file f compared with other records of the same type. Specifically, $f_{r,f}$ is the count of r in f. For example, assume a file f contains two API invocations m_1 and m_2; $f_{m_1,f}$ is 1 and $\sum_{r' \in f} f_{r',f}$ is 2.

F is the universal set of files in a CPG G. $IDF(r,F)$ is the *inverse document frequency* for r. It evaluates how rare is r. Suppose the value for $|F|$ is 10000. Only two files contain m_1 and fifty files contains m_2. $IDF(m_1,F)$ is 5000 and $IDF(m_2,F)$ is 200.

Take the product of these two, $TF\text{-}IDF(r,f)$ indicates the importance of r for f, which is r's importance value as well. $TF\text{-}IDF(m_1,f)$ is hence 25 times of $TF\text{-}IDF(m_2,f)$.

Assigning Weights to CPG Edges. Each edge in a weighted CPG or WCPG has a weight in the range of (0,1]. Data flow or control edges always have a weight value of 1, meaning



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

Figure 7: Weight Propagation

that they are always important and hence the attack inspection tool shall traverse them. For undirected match edges, ARTISAN assigns some initial weights that are smaller than 1.0. Intuitively, an edge has a large weight if the two files involved in the edge share similar important UFM records. For example, the match edge between f_1 and f_2 has the largest edge weight if both files share the same set of important records. The formal definition is the following.

$$MatchWeight(f_1,f_2) = \frac{\sum_{r\in f_1\cap f_2} TF\text{-}IDF(r,f_1)+TF\text{-}IDF(r,f_2)}{\sum_{r_1\in f_1} TF\text{-}IDF(r_1,f_1)+\sum_{r_2\in f_2} TF\text{-}IDF(r_2,f_2)}$$

The weight of a match edge between two files is calculate by the importance of their overlapping records divided by the importance of all the records in the two files.

Example. Assume $f_1=\{r_1,r_2\}$, $f_2=\{r_1\}$, and $f_3=\{r_2\}$, $|F|=10000$, two files containing r_1 and fifty files containing r_2. $TF\text{-}IDF(r_1,f_1)=TF(r_1,f_1)*IDF(r_1,F)=0.5*5000=2500$. Similarly, we can get $TF\text{-}IDF(r_2,f_1)=100$, $TF\text{-}IDF(r_1,f_2)=5000$, $TF\text{-}IDF(r_2,f_3)=200$. We hence have $MatchWeight(f_1,f_2)=0.98$ and $MatchWeight(f_1,f_3)=0.1$ following the above formula. Intuitively, f_1 and f_2 are very similar as only these two files share some unique record r_1. In contrast, although f_1 and f_3 share r_2, they do not have much similarity as r_2 is present in a large number of files.

Computing Weights for CPG Nodes. Different from edge weights that denote the confidence of correlations, node weights in CPG denote the likelihood that nodes are part of the attack. Such likelihood is determined by not only correlation weights, but also connections to attack symptoms. Intuitively, assume a node f_2 is known to be part of the attack and there is an edge between f_1 and f_2. If the edge weight is small, the likelihood of f_1 being part of the attack is low. On the other hand, even if an edge between f_1 and f_2 has a large weight, if neither node is part of the attack, the edge is not part of the attack. Therefore, node weights are computed by starting from an attack symptom node, which has the largest weight 1.0, and traversing the CPG along the important edges.

Algorithm 1 presents the node weight computation procedure. At the beginning, undirected match edges $undirectedEdge(u,v)$ in CPG are first replaced with two directed edges $directedEdge(u,v)$ and $directedEdge(v,u)$ (lines 1-4). For a directed edge $directedEdge(u,v)$, we normalize its weight by dividing the sum of weights of all outgoing edges from the source node u (lines 5-6). The weight computation procedure is iterative, guarding by a convergence threshold. It means that the computation terminates if the maximum change in one round of computation is smaller than the threshold. We use $\tau_c = 1e-8$ as it gives robust results from our evaluation. In each iteration, the weight of each node (except the symptom node) is the weighted sum of its child nodes' weights (line 11-14). Finally, ARTISAN only reports the subgraph that has nodes with weights larger than $\tau_d = 0.05$, i.e., the ranked provenance graph.

Algorithm 1: Get Ranked Provenance Graph

Input: Weighted Provenance Graph G_w,
Symptom Node s,
Convergence Threshold τ_c,
Weight Threshold τ_d
Output: Ranked Graph $G_{w'}$

```
1  for ∀ undirectedEdge(u, v) ∈ G_w do
2      G_w ← G_w \ undirectedEdge(u,v)
3      G_w ← G_w ∪ directedEdge(u,v)
4      G_w ← G_w ∪ directedEdge(v,u)
5  for ∀ u ∈ G_w do
6      directedEdge(u,v).weight =
           directedEdge(u,v).weight / Σ_{directedEdge(u,v')} directedEdge(u,v').weight
7  s.weight ← 1
8  while change > τ_c do
9      for ∀ u ∈ G_w do
10         if u ≠ s then
11             value ← 0
12             for ∀ v ∈ directedEdge(u, v) do
13                 value += v.weight *
                       directedEdge(u,v).weight
14             u.weight ← value
15 rank nodes in G_w' based on their node weights
16 return G_w' with nodes weights larger than τ_d
```

Example. Figure 7(a) shows the CPG with edge weights for the previous example. Assume file f_1 is detected as an attack symptom. Recall that the undirected edges are match edges and the directed edge is a dataflow edge. The match edge between f_1 and f_2 can be replaced by two directed edge (f_1, f_2) and (f_2, f_1). We hence get Figure 7(b). The sum of weights of all outgoing edges from f_1 is greater than 1. Hence we need to normalize it to ensure the sum equals to 1.0. We hence get Figure 7(c). The weight for the symptom node is 1.0 to begin with and will not be changed during the computation. For a node f, its weight is the weighted sum (using normalized edge weights) of the node weights of all parents. In Figure 7(d), the weight for node f_2 is the edge weight 0.47 multiplied by the node weight of its parent f_1. Hence the weight of f_2 is 0.47. The node weights are iteratively updated till convergence.

Table 3: Attack overview

No.	Device	Duration	Scenario Name	Attack Reference
1	NextCloud Box	1d0h	SMTP Command Injection	CVE-2022-24838
2	NextCloud Box	0d23h	Attachment Header Injection	CVE-2021-32679
3	NextCloud Box	0d19h	Configuration Attack	CVE-2021-32654
4	NextCloud Box	0d3h	Command Injection	CVE-2019-12739
5	NextCloud Box	0d10h	Code Injection	CVE-2020-8180
6	OpenHAB Hub	0d8h	XXE Attack	CVE-2021-21266
7	OpenHAB Hub	0d3h	Binding Attack	CVE-2020-5242
8	OpenHAB Hub	0d20h	Credential Leak	CVE-2021-33024
9	OpenHAB Hub	0d10h	Denial-of-Service	CVE-2018-7580
10	OpenHAB Hub	1d23h	Password Cracking	CVE-2021-33020
11	Google Hub	1d2h	Unauthorized Access	CVE-2020-24441
12	Google Hub	0d8h	Cross Site Scripting	CVE-2021-29953
13	Android Phone	0d23h	RCE Attack	CVE-2021-40724
14	Android Phone	1d0h	HTML Injection	CVE-2021-29944
15	Personal Computer	0d5h	Phishing Email Link	TC [5] Case 4.5
16	Personal Computer	1d23h	Phishing Email Exec	TC [5] Case 4.8

4 Evaluation

The evaluation addresses the following research questions:
RQ1: What is the logging overhead of ARTISAN, including both space and runtime (Section 4.2)?
RQ2: What is the efficiency of weighted content provenance graph construction (Section 4.3)?
RQ3: What is ARTISAN's effectiveness in attack forensics and how does it compare to existing techniques (i.e., ALchemist, CLARION and eAudit), with respect to the precision, recall, and efficiency (Section 4.4)?

4.1 Experiment Setup

Environment. The experiments are conducted on five different devices, including three IoT hubs, a mobile phone, and a personal computer. For IoT hubs, we choose three mainstream smart home platforms, i.e., a Nextcloud VM Appliance [13] running on Raspberry equipped with a 950MHz ARM Cortex-A53 CPU and 1G RAM, an OpenHAB Hub on Raspberry equipped with a 1GHz ARM Cortex-A57 CPU and 512MB RAM, and an emulated Google Hub running on Raspberry equipped with a 1.2 GHz quad-core ARM Cortex-A53 and 2G RAM. They are connected with several home devices (e.g., Philip lighting device). In addition to IoT devices, ARTISAN can also be used in mobile phone attack forensics, in which the resource constraints are similar. We hence include an Android phone with a 2.0GHz Qualcomm Snapdragon and 4GB RAM. Although ARTISAN is designed for IoT devices or computing systems with a small budget for logging, to further show the generality of ARTISAN and compare with baselines, we also include a laptop equipped with an Intel i7-9700 CPU 4.7GHz, 64GB memory, and an up-to-date Ubuntu 20.04. All these systems are continuously run for a month. Within the period, these systems handle regular benign workloads, and we occasionally launch remote attacks from different attacker IPs. That is, the systems are dealing with normal workloads, e.g., as the primary machine/phone for daily usage, in most of the time.

To support ARTISAN, the ZFS system is deployed on all five devices. To support eAudit, the BCC toolkit [3] is deployed on all devices as well. These devices are used exclusively by the authors of this paper. Before using the collected data, we anonymize the identity related information including user accounts, private file names, and private domain names.

Attack Selection and Ground Truth. To faithfully reproduce IoT attacks, we searched all the attacks in recent 3 years with detailed steps and reproducible PoCs. In Table 3, we covered a broad range of IoT attack types. For the attacks on PCs, we downloaded the ground-truth reports with descriptions of attack steps from the DARPA TC dataset. We strictly followed these steps to reproduce the attacks. We followed the literature [5] to set up the experiments and compare with the baselines. Similar to previous work [75], we understood the attack workflow, and manually labeled nodes and edges in all the CPGs and APGs as the ground truth, considering their relations to attacks.

Baselines and Fair Comparison. We chose CLARION and ALchemist as they are the most recent work on whole system provenance. Both systems are built upon SPADE [35] for data collection and management. They implemented new components to collect additional provenance events (e.g., application events for ALchemist) and integrated the events into APGs. For a fair comparison, the same symptom event is provided, and we strictly followed their settings to obtain the final APGs. Both works do not require instrumentation and are thus applicable to various Linux based devices including Android, IoT Hub and PC. Existing IoT provenance systems (e.g., ProvThing [66] for SmartThings platform) instrument platform applications to collect provenance information. Hence they require intensive manual efforts to instrument applications when deploying on different platforms. Besides, there is a recent work eAudit which leverages eBPF framework to develop an efficient data collection system. However, our system is different from eAudit. Our system records information in the spatial dimension whereas eAudit records information in the temporal dimension. Besides, we had optimized the configuration of the underlying Linux audit system (on which CLARION and ALchemist ran) and eAudit system, in order for them to run efficiently on the IoT devices. For each device, we ran the audit subsystem and eAudit system with different workloads to collect statistics (e.g., backlog, rate and event losses). These statistics were used to optimize configurations (e.g., buffer size). Our goal was to achieve optimal performance without losing events, ensuring the tools continued collecting the intended data. Therefore, we carefully optimized the configuration (of linux audit system) to maintain the original tools' effectiveness. Note that for efficiency, these tools had already optimally limited the types of recorded syscalls to only those forensic related. Customizing their settings might be impractical (e.g., SPADE cannot capture file

Table 4: Space overhead (one week)

System	Applications	NextCloud Box		OpenHAB Hub		Google Hub		Android Phone		Personal Computer	
		Items(#)	Space(MB)	Items(#)	Space(MB)	Items(#)	Space(MB)	Items(#)	Space(MB)	Items(#)	Space(MB)
ARTISAN	Overall	1560531	580.17	2143183	796.55	1781169	687.90	686104	265.33	6197367	3456.04
	Browser	1092373	394.51	1455459	540.57	1295851	490.69	510019	189.28	4340488	2350.72
	PDF Reader	815	17.40	1092	23.32	733	16.28	631	13.99	1965	41.97
	Video Player	268	9.85	275	10.14	330	12.16	202	7.38	412	15.21
	Mail	2484	46.41	3247	62.23	1676	32.48	1124	21.78	2548	44.88
	Chat	-	-	-	-	-	-	296	11.07	917	34.31
eAudit	Overall	30324995	1921.52	44272206	2810.54	32973105	2092.75	11650230	1042.66	95025743	8504.38
	Browser	22683096	1433.58	32785379	2143.34	22402571	1424.34	7932071	709.90	54969114	4914.63
	PDF Reader	454809	41.48	619810	53.88	495515	44.86	175284	15.86	1153979	103.16
	Video Player	900521	56.90	796898	72.66	793700	71.84	280808	25.42	1846366	170.68
	Mail	1411727	89.22	1155266	85.80	621944	60.41	338906	41.30	2161962	199.85
	Chat	-	-	-	-	-	-	822165	100.64	5420404	450.83
CLARION	Overall	69747489	11303.44	94283403	17472.65	75837495	13236.76	18976552	3312.44	154783293	73113.60
	Browser	47428292	7686.04	61284212	11352.68	51504538	9540.46	9865121	1722.46	90984138	42977.01
	PDF Reader	1046212	169.54	1319967	244.60	1136218	210.55	424623	74.11	1731796	818.02
	Video Player	2071500	339.09	1697101	314.49	1806680	334.27	1192896	208.19	2434246	1149.84
	Mail	3347879	542.25	2451368	454.27	1285024	237.99	1444076	252.03	9423088	4451.15
	Chat	-	-	-	-	-	-	1978557	345.31	10759276	5082.26
ALchemist	Overall	20232268	3229.85	26135359	4893.56	20788919	3892.05	9109955	1589.97	34861730	16806.71
	Browser	13757870	2195.78	19019638	3497.22	1530365	2865.03	5209476	892.05	20917038	10084.02
	PDF Reader	303334	48.27	373200	69.86	327620	61.32	207586	36.28	390038	190.87
	Video Player	594828	94.86	479836	89.83	580790	107.72	583487	101.77	547866	274.33
	Mail	970173	154.83	693106	129.76	413582	78.17	706917	123.85	2334557	1023.12
	Chat	-	-	-	-	-	-	967804	170.46	2423198	1168.16

content as we do) or suboptimal (e.g., potentially missing necessary information). We hence refrained from altering their default settings. We acquire the implementations of baseline techniques from their authors and follow the original settings to set up the environment. Specifically, to enable CLARION and ALchemist, we deploy the audit logging systems on five devices. To support eAudit, BCC toolkit [3] is deployed on all five devices. We additionally enable the application logging modules for ALchemist.

Provenance Data. CLARION, ALchemist, and eAudit captures the syscall interactions between system subjects (e.g., process) and system objects (e.g., file or network socket). The CLARION captures all syscall types while eAudit only captures 80 syscalls and ALchemist only captures 66 syscalls. Besides, ALchemist also record the interactions between application internal structures (e.g., tabs in firefox) and system subjects/objects. In contrast, ARTISAN record forensic related content changes between file system snapshots.

4.2 RQ1: System Overhead

Affordable runtime and space overhead is crucial for attack provenance systems since they usually operate in a long-term setting and are active during users' daily uses. In this section, we study the overhead of ARTISAN and compare it with that of baseline techniques. To measure the runtime overhead, we use the *postmark* benchmark [48], a file system benchmark that simulates the behavior of a mail server. Note that this benchmark is widely used by existing works [28, 61]. Following eAudit's configuration, we set it to run 4500 transactions with file sizes ranging from 4KB to 50KB across 10 sub-directories. We obtain the configurations from eAudit and run it on five devices. Similar to many existing work [28,59,61], when evaluating the time overhead, we run the same workload with/without ARTISAN/eAudit/ALCHEMIST/CLARION. The overhead is calculated as the ratio of the CPU time used by the provenance collection system to that of the benchmark. To measure the space overhead, we measure the additional auxiliary files created during a one-week period (although the systems run for a month). Specifically, we collect file change records for ARTISAN, system logs for eAudit and CLARION, and both system logs and application logs for ALchemist. Table 4 presents the space overhead. The first column denotes the forensic techniques and the second column the subject applications. Columns 3-4, 5-6, 7-8, 9-10, and 11-12 denote the space overhead on the five testing devices, respectively, where we first present the number of log items and then the consumed space. Items mean the number of file changes for ARTISAN, and the number of log entries for the three baselines. The "overall" rows indicate the total space usage and the other rows list the dominating applications' space consumption. Observe that ARTISAN is the most space-friendly. Specifically, it consumes around 1157MB on average (on the five devices), In contrast, eAudit consumes roughly three times more resources, while CLARION and ALchemist require about six to twenty-four times more resources. With further inspection, we find that ARTISAN performs much better on applications with large workloads, e.g., web browsers. This is mainly because that ARTISAN is file-based and does not need to record a sheer quantity of system events.

Table 5: Runtime overhead

System	Nextcloud Box	OpenHAB Hub	Google Hub	Android Phone	Personal Computer
ARTISAN	3.82%	3.98%	3.41%	3.67%	3.54%
eAudit	5.58%	5.86%	5.34%	5.60%	5.16%
CLARION	45.84%	48.46%	46.38%	44.25%	42.68%
ALchemist	43.96%	45.81%	44.61%	42.13%	40.54%

Table 5 depicts the runtime overhead. On average, the runtime overhead of ARTISAN is 3.68%, smaller than the state-of-the-art eBPF based provenance system eAudit (5.50%). The overhead for two auditd based baselines CLARION and ALchemist are 45.52% and 43.41%, respectively. The underlying reason is that the runtime overhead of ARTISAN originates from taking the ZFS snapshots, which is efficient by design. In comparison, CLARION and ALchemist need to frequently record file read/write system events. The eAudit also incurs small overhead due to its efficient encoding method that reduces the cost of user-kernel communication within the eBPF framework. We also observe that CLARION has larger overhead compared to ALchemist. It is because CLARION needs to record and process all kinds of audit log while ALchemist only focuses on a selected subset (66 syscalls). The results are consistent with the reported results in [61].

4.3 RQ2: Efficiency of CPG Construction

The overhead of CPG construction are dominated by the datalog inference time and weight propagation. Recall that ARTISAN is demand-driven and only performs inference on atoms related to attacks. Table 6 presents the statistics of datalog inference. The first column shows the attacks. The second column shows that how many atoms are processed, without and with demand-driven analysis. For instance, 65.6K/10.8K (1st row) means that without demand-driven, 65.6K atoms have to be processed, and with demand-driven, they are reduced to 10.8K. The third column reports the number of applications of inference rules (without and with demand-driven). The fourth column shows the number of derived relations; the fifth column time consumed and the last column memory consumed. The results indicate the necessity of the demand-driven strategy. Observe that in the complex attack 15 (involving complex *Firefox* behaviors), the inference engine applies over 400 thousand rules, deriving 128 thousand new relations. The corresponding runtime overhead is only 21.7 seconds while the space overhead is only 166MB, demonstrating the practicality of ARTISAN in attack forensics.

The weight propagation time is shown in the 7th column in Table 6. On average, it takes 612.5 seconds to propagate weights. Note that in the most complex attack 15, the corresponding runtime overhead is less than one hour, demonstrating the practicality of ARTISAN.

Table 6: Datalog inference details of attacks

Attack	Atoms(#)	Rules(#)	Relations(#)	Time(s)	Memory(MB)
1	65.6K/10.8K	733.5K/31.8K	163.0K/11.9K	19.8 / 0.5	78 / 12
2	93.1K/16.5K	1.27M/73.4K	302.6K/48.8K	67.1 / 3.5	138 / 35
3	36.2K/5.9K	411.5K/17.9K	85.8K/6.6K	11.4 / 0.3	43 / 7
4	51.0K/8.3K	693.3K/34.6K	160.4K/27.2K	25.0 / 1.9	75 / 19
5	35.8K/6.8K	409.7K/17.1K	87.5K/6.5K	10.3 / 0.3	52 / 16
6	26.5K/4.1K	356.8K/13.4K	72.6K/5.9K	16.7 / 0.4	88 / 4
7	167.4K/28.0K	1.9M/81.2K	423.7K/31.0K	148.9 / 5.6	298 / 60
8	113.8K/20.3K	1.2M/52.5K	291.4K/21.3K	135.2 / 5.0	258 / 42
9	31.9K/5.4K	367.6K/15.1K	79.3K/5.6K	10.9 / 0.3	39 / 6
10	146.7K/23.2K	1.7M/67.8K	370.6K/26.2K	146.5 / 5.2	307 / 67
11	49.2K/8.8K	533.3K/24.0K	123.2K/9.1K	14.2 / 0.4	61 / 10
12	124.5K/19.7K	1.4M/58.3K	296.3K/21.6K	124.8 / 4.1	280 / 46
13	16.4K/2.5K	188.2K/8.0K	41.9K/3.1K	4.8 / 0.1	18 / 3
14	12.3K/1.8K	140.9K/6.0K	29.1K/2.2K	3.7 / 0.1	15 / 2
15	425.6K/71.4K	5.6M/408.2K	1.3M/128.5K	644.8 / 21.7	1288 / 166
16	502.8K/97.6K	5.3M/240.5K	1.2M/94.2K	632.5 / 16.4	1165 / 135

4.4 RQ3: Effectiveness in Attack Forensics

We evaluate the effectiveness of ARTISAN in real-world attack provenance. Specifically, we utilize ARTISAN and the three baselines to construct provenance graphs and compare them with the ground truth. Note that eAudit, ALchemist and CLARION construct attack provenance graphs (APGs), which are composed of system events instead of files and URI nodes in CPGs by ARTISAN. To make a fair comparison, we project all the network event nodes in APGs to their corresponding URIs and non-network events to the corresponding file nodes. Specifically, process nodes in APGs are projected to process file nodes. Process creation edges are projected to control edges (in CPGs). File reads and writes in APGs are directly translated to CPG file nodes and data flow edges. After translation, we compute the precision and recall of graphs. Table 7 presents the forensic results. The first column presents the attack ids. Columns 2-3 show the ground truth, where column 2 denotes the number of attack relevant nodes and column 3 the normal nodes. Note that the benign nodes are dominant, making attack forensics challenging. Columns 4-7, 8-11, and 12-15 present the results of ARTISAN, ALchemist, and CLARION, respectively. We present the number of false positives (FP), the number of false negatives (FN), precision, and recall. Precision is defined as the percentage of nodes in the APG/CPG that are related to the attack. Recall is defined as the percentage of attack nodes that are covered in the APG/CPG. For example, a precision of 80% indicates 80% percent of nodes in final APG/CPG are relevant to attack. A recall of 80% indicates 80% of the attack nodes are contained in APG/CPG. Observe that ARTISAN performs the best in terms of precision (93.9%) and recall (98.8%). It is worth noting that ARTISAN achieves 100% recall for the first 14 attacks on IoT devices, indicating all the attack nodes are successfully identified by ARTISAN, while ALchemist, CLARION and eAudit miss a few. ARTISAN also outperforms the baseline techniques in term of precision, for the first 14 attacks, indicating ARTISAN involves fewer attack-irrelevant nodes in the graph. The two attacks (#15 and #16) from the DARPA

Table 7: Forensic results (in |File|)

Attack No.	Ground Truth		ARTISAN				ALchemist				CLARION				eAudit			
	Attack	Normal	FP	FN	Precision	Recall	FP	FN	Precision	Recall	FP	FN	Precision	Recall	FP	FN	Precision	Recall
1	19	23532	2	0	90.5%	100%	16	0	54.3%	100%	24	0	44.2%	100%	24	0	44.2%	100%
2	30	40724	3	0	90.9%	100%	22	0	57.7%	100%	28	0	51.7%	100%	28	0	51.7%	100%
3	9	16276	0	0	100%	100%	8	2	52.9%	81.8%	8	2	52.9%	81.8%	8	2	52.9%	81.8%
4	70	20947	4	0	94.6%	100%	27	0	72.2%	100%	39	0	64.2%	100%	39	0	64.2%	100%
5	15	15143	1	0	93.8%	100%	10	3	60.0%	83.3%	12	3	55.6%	83.3%	12	3	55.6%	83.3%
6	17	11635	0	0	100.0%	100%	7	5	70.8%	77.3%	11	5	60.7%	77.3%	11	5	60.7%	77.3%
7	50	67537	7	0	87.7%	100%	34	2	59.5%	96.2%	42	3	54.3%	94.3%	42	3	54.3%	94.3%
8	40	47483	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%
9	24	14051	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%
10	43	59250	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%	0	0	100%	100%
11	28	19872	3	0	90.3%	100%	6	0	82.3%	100%	6	0	82.3%	100%	6	0	82.3%	100%
12	45	54127	6	0	88.2%	100%	14	0	76.2%	100%	17	0	72.5%	100%	17	0	72.5%	100%
13	31	6961	2	0	93.9%	100%	7	0	81.6%	100%	10	0	75.6%	100%	10	0	75.6%	100%
14	34	5138	0	0	100%	100%	5	0	87.2%	100%	5	0	87.2%	100%	5	0	87.2%	100%
15	55	174142	8	6	87.3%	90.2%	0	4	100%	93.2%	0	4	100%	93.2%	0	2	100%	96.5%
16	62	205303	10	6	86.1%	91.2%	0	4	100%	93.9%	0	4	100%	93.9%	0	2	100%	96.8%
Avg.	35	48882	3	0	93.9%	98.8%	10	1	78.4%	95.3%	18	2	75.1%	95.2%	18	2	75.1%	95.6%

TC engagement aim to escalate privileges after gaining access to the victim's computer. Subsequently, they attempt to compromise running applications such as *sshd* to maintain persistence. ARTISAN loses information about in-memory process injection, resulting in lower precision and recall compared to the baseline. In such scenarios, the log tampering window is crucial. eAudit implements a smaller tampering window, allowing more provenance data to be collected before the audit is turned off. Therefore, eAudit achieves better results compared to ALchemist and CLARION.

Why our result is better than baselines. With further inspection, we observe that these attacks exploit vulnerabilities (e.g., alter configuration files) to perform malicious behaviors such as delivering payload, executing malicious commands or exposing sensitive data. And we observe that the functionalities involved in attacks are provenance ready. ARTISAN hence captures all the necessary information. In contrast to ARTISAN, both eAudit, CLARION and ALchemist might miss nodes and edges introduced by control flow (e.g., permission changes in Section 2) and dataflow. As demonstrated in Section 2, they lack an understanding of the application-level semantics. For example, in attack #5, ARTISAN can infer the dataflow edge from attack's IP to local file by analyzing API invocation record from *Talk*, while eAudit, CLARION and ALchemist missed such edges. Besides, eAudit, CLARION and ALchemist cannot distinguish between internet connections triggered by APIs from input files and those from regular application server connections or benign activities (e.g., chatting with friends in attack #5). Consequently, many benign connections are mistakenly considered malicious without understanding the contents of application input files.

5 Discussion and Limitations

Obfuscation. As mentioned in Section 3.2, the efficacy of ARTISAN hinges on the robustness of the file parsing and normalizing processes, which are susceptible to adversarial interference by obfuscating the file content. It is important to note that these obfuscation techniques are not challenges unique to ARTISAN, but are indeed prevalent issues faced by existing techniques [38, 56, 71, 74]. Specifically, methods of dynamic obfuscation [49] possess the capability to inject numerous redundant behaviors during runtime. This could result in a high number of false positives for existing works, rendering the provenance graph meaningless. We further confirmed with the authors of UIScope [71] and ALchemist [74] that their systems can also be adversely impacted by obfuscation methods. In addressing obfuscated files, ARTISAN may potentially overestimate the files as being related to all others, or conversely, underestimate their relevance, depending on the analytical context. We argue the task of de-obfuscation is orthogonal to our work, and, thus, leave it as future works.

Fileless Attacks in IoT Context. ARTISAN cannot handle attack steps that do not leave trails in the file system, namely fileless attacks. One typical failure case involves attacks using inter-process communication (IPC). However, IPC is less of a problem for IoT devices because they enforce strict access controls, preventing a process from an application from directly sending messages to another process or reading/writing files possessed by another process by default [40]. Although attackers can exploit vulnerabilities to add malicious processes to the trust list of the target application process for future intrusions, such manipulation is recorded in the access configuration file of the target application. Thus, the initial intrusion can still be identified. On the other hand, attackers could use evasion techniques like memory injection and launch adaptive attacks to intentionally evade forensic tracking by ARTISAN. Note that directly tampering with other processes' memory usually requires strong permissions, which could allow an attacker to corrupt most audit systems. That is, most existing forensic tracking techniques [27, 61, 74] share this limitation and would fall short of these evasion techniques.

Solving this problem may require finding a Trusted Computing Base (TCB) in the lower part of the system stack, such as hardware, which is orthogonal to ARTISAN. We hence leave it as future work.

One-minute Interval. ARTISAN relies on file system snapshots, which means attackers may evade detection by hiding their activities within a one-minute interval. For example, an attacker could frequently rewrite the payload file within this interval, making the changes invisible to ARTISAN. To understand the prevalence of such attacks in real world, we test our system on commonly used attack dataset including DARPA TC [5] and previous literature [38, 60, 74], we find out that typical attacks do not involve such behavior. Note that in this dataset, some attacks can indeed delete the original payload within the snapshot interval. However, our system can still capture these attacks. This is because we set the *nounlink* attribute in ZFS (Section 3.2), which ensures that when a file is deleted, ZFS retains the file content. Nevertheless, as previously mentioned, attackers can potentially launch adaptive attacks to exploit the one-minute interval. We leave the detection of adaptive attacks within the one-minute interval as future work, as adaptive attacks remain a threat to all forensic tracing techniques.

Extensibility and Adaptability to New File Types. Integrating new file types into ARTISAN primarily necessitates that analysts provides a parser to translate file contents into UFM, a task that generally constitutes a one-time effort. For many file types, existing analysis tools are available, making the integration process relatively straightforward and not overly time-consuming. For example, employing the Office Analyzer [14] enabled a senior analyst to integrate Microsoft Excel into ARTISAN within approximately two hours. Conversely, for file types that lack publicly available analysis tools, especially proprietary formats, the integration process may be more time-consuming. Implementing deep-learning-based methods has the potential to reduce these efforts, and we plan to explore this approach in our future work.

Limited Set of Rules. Given new systems and attacks, we need to study if the applications and file types involved in the attacks are content-forensic ready (Section 3.1). If so, CPG is equally capable as APG. If not, ARTISAN may conservatively flag them as all related to attacks.

6 Related Work

IoT Security. There has been an increasing amount of research [22–24, 46, 64–66] on IoT security and more broadly IoT safety. These works mostly focus on vulnerability detection, integrity protection, and authentication. ProvThings [66] instruments security-sensitive application APIs to track data assignments and method invocations. ARTISAN does not require instrumentation, and considers both the application and file type APIs to build up system-level data provenance.

Provenance Collection. Various systems have been developed to collect provenance data. Audit log-based systems such as Spade [35] and Trace [44] utilize existing logging mechanisms to gather provenance information. In contrast, some approaches involve direct instrumentation of the OS kernel to enhance the granularity and security of the data collected. Examples of such systems include LPM [20], HiFi [60] and CamFlow [59]. PROVBPF [54] and eAudit [61] both leverage the Linux eBPF framework, with the former adapting CamFlow's methods for efficient provenance in container environments, and the latter introducing novel encoding techniques that reduce communication overhead, enabling the handling of significantly heavier workloads. While these systems primarily focus on tracking and analyzing provenance from a temporal perspective, our work focuses on the collection and analysis of the spatial dimension of provenance data.

Forensics analysis. Causality analysis played a critical role in forensics analysis [55, 59, 60]. When an attack symptom is identified, the analyst can utilize the provenance data to perform backward and forward tracking to identify the root cause and ramifications. Existing works focus on addressing problems such as removing redundant events and causal relations [38, 42, 52, 56, 74], reducing logging overhead [57, 60], developing query system [32–34, 55], graph compression [68, 69], and semantic pruning [29, 42, 63, 70].

7 Conclusion

We propose a novel cost-effective attack forensics technique that is based on spatial logs instead of events in the temporal dimension. It features a novel representation called content provenance graph, which is equally informative as traditional attack provenance graphs. Our technique is much more efficient than the state-of-the-arts while having better precision and recall in IoT attack forensics.

References

[1] 56 email statistics. https://financesonline.com/email-statistics/.

[2] Androguard. https://github.com/androguard/androguard.

[3] Bcc. https://github.com/iovisor/bcc.git.

[4] Cve-2021-32654. https://nvd.nist.gov/vuln/detail/CVE-2021-32654.

[5] Darpa transparent computing program. https://github.com/darpa-i2o/Transparent-Computing.

[6] Exiftool. https://exiftool.org/.

[7] Frida. https://frida.re/.

[8] Ghidra. https://en.wikipedia.org/wiki/Ghidra.

[9] Google play app store. https://play.google.com/store/apps.

[10] Kernel patch protection. https://en.wikipedia.org/wiki/Kernel_Patch_Protection.

[11] Most popular file system. https://tinyurl.com/mr2w6u8p.

[12] Nextcloud box applications. https://github.com/orgs/nextcloud/repositories.

[13] nextcloud-vm. https://www.hanssonit.se/nextcloud-vm/.

[14] Opendocument analyzer. https://github.com/dstosberg/odt2txt.

[15] Sanoid. https://github.com/jimsalterjrs/sanoid/.

[16] Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis.

[17] Virustotal report. https://assets.virustotal.com/reports/2021trends.pdf.

[18] Zfs. https://docs.oracle.com/cd/E19253-01/819-5461/zfsover-1/index.html.

[19] Zfs license. https://openzfs.github.io/openzfs-docs/License.html.

[20] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security*, 2015.

[21] Brian Carrier. *File system forensic analysis*. Addison-Wesley Professional, 2005.

[22] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *USENIX Security*, 2018.

[23] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *USENIX ATC*, 2018.

[24] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *NDSS*, 2019.

[25] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*.

[26] Ping Chen, Lieven Desmet, and Christophe Huygens. A study on advanced persistent threats. In *CMS*, 2014.

[27] Xutong Chen, Hassaan Irshad, Yan Chen, Ashish Gehani, and Vinod Yegneswaran. Clarion: Sound and clear provenance tracking for microservice deployments. In *USENIX Security*, 2021.

[28] Feng Dong, Shaofei Li, Peng Jiang, Ding Li, Haoyu Wang, Liangyi Huang, Xusheng Xiao, Jiedong Chen, Xiapu Luo, Yao Guo, et al. Are we there yet? an industrial viewpoint on provenance-based endpoint detection and response tools. In *CCS*, 2023.

[29] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. Seal: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *USENIX Security*, 2021.

[30] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *S&P*, 2016.

[31] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *USENIX Security*, 2016.

[32] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Haoyuan Liu, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. A system for efficiently hunting for cyber threats in computer systems using threat intelligence. In *ICDE*, 2021.

[33] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. Saql: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security*, 2018.

[34] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. Aiql: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, 2018.

[35] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2012.

[36] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*, 2018.

[37] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *S&P*, 2020.

[38] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[39] Wajih Ul Hassan, Mohammad Ali Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *NDSS*, 2020.

[40] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (iot). In *USENIX Security*, 2018.

[41] Dominique A Heger. Workload dependent performance evaluation of the btrfs and zfs filesystems. In *Int. CMG Conference*, 2009.

[42] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *USENIX Security*, 2017.

[43] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *S&P*, 2020.

[44] Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Hyung Lee, Jignesh Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. Trace: Enterprise-wide provenance tracking for real-time apt detection. *TIFS*, 2021.

[45] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security*, 2018.

[46] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, and Atul Prakash. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, 2017.

[47] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. 2016.

[48] Jeffrey Katcher. Postmark: A new file system benchmark. *TR3022*, 1997.

[49] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled ElWazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic obfuscation in malware. In *DIMVA*, 2017.

[50] Samuel T King and Peter M Chen. Backtracking intrusions. *SOSP*, 2003.

[51] William Largent. New vpnfilter malware targets at least 500k networking devices worldwide. https://blog.talosintelligence.com/vpnfilter/, 2018.

[52] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[53] Bo Li, Phani Vadrevu, Kyu Hyung Lee, Roberto Perdisci, Jienan Liu, Babak Rahbarinia, Kang Li, and Manos Antonakakis. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.

[54] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure namespaced kernel audit for containers. In *SoCC*, 2021.

[55] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[56] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Mpi: Multiple perspective attack investigation with semantics aware execution partitioning. In *USENIX Security*, 2017.

[57] Shiqing Ma, Xiangyu Zhang, Dongyan Xu, et al. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[58] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *ATEC*, 2006.

[59] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *CCS*, 2018.

[60] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC*, 2012.

[61] R Sekar, Hanke Kimm, and Rohit Aich. eaudit: A fast, scalable and deployable audit data collection system. In *S&P*, 2023.

[62] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 2001.

[63] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *CCS*, 2018.

[64] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *USENIX Security*, 2017.

[65] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action iot platforms. In *CCS*, 2019.

[66] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *NDSS*, 2018.

[67] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, 2005.

[68] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell DE Long. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS)*, 2013.

[69] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell DE Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. Compressing provenance graphs. In *TaPP 11*, 2011.

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

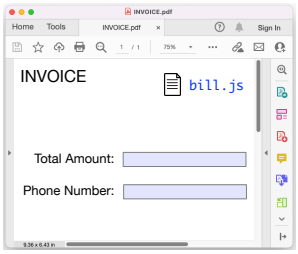
[71] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *NDSS*, 2020.

[72] Attila A Yavuz, Peng Ning, and Michael K Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *FC*, 2012.

[73] Attila Altay Yavuz and Peng Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *ACSAC*, 2009.

[74] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, et al. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *NDSS*, 2021.

[75] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *S&P*, 2022.



(a) INVOICE.pdf

```
1  DataSource: add0n.com/pdf.html
2
3  Object 1  << /Type /Catalog >>
4  /OpenAction /Launch /F bill.js
5
6  Object 2
7  << /Type /EmbeddedFile bill.js >>
8  ...
9  app.launchURL(http://nato.com/
10 payload.php)
11
12 Object 3 <</Type /AA>>
13 /Keystroke /SubmitForm
14 /URL (http://nato.com/upload.php)
```

(b) source of INVOICE.pdf

Figure 8: A PDF file and its source

Appendix

A UFM Abstract Types: A PDF Example

PDF is one of the most widely used types of document. It has a large number of advanced features (e.g., 3D animations) and allows embedding different kinds of objects (e.g., Flash) and script code blobs (e.g., Javascript). While these embedded objects and script code snippets provide exceptional expressiveness and functionalities, they may have severe security problems. Based on a recent report [17], thousands of PDF CVEs are originating from these auxiliary features. For example, many PDF malwares leverage JavaScript APIs.

Figure 8(a) shows a malicious PDF sample *INVOICE.pdf*. It pretends to be a benign invoice file and deceives the user to enter sensitive information. When the file is opened, the malicious code in embedded file *bill.js* will be executed. It loads a remote URL *http://nato.com/payload.php*. Besides, when it detects keystrokes, which contain sensitive information, it sends the keystrokes through a form to a remote URL *http://nato.com/upload.php*. Figure 8(b) shows a part of the internals of *INVOICE.pdf*. In particular, line 1 specifies the data source (*add0n.com/pdf.html*). Lines 3-4 show a *Catalog* object which is the root object for a PDF file. The *OpenAction* label (line 4) indicates an open event (of the PDF file) and *Launch* indicates the JS file invoked. The two lines mean that when the PDF file is opened, it automatically launches the embedded file *bill.js*, which is defined in lines 7-10. It contains an API invocation to *launchURL* to load *http://nato.com/payload.php*. Lines 12-14 define an additional action: when a keystroke event is detected, it will submit a form to *http://nato.com/upload.php*.

Line 1 in Figure 8(b) is parsed to a metadata record. All the file paths (*bill.js*) and URLs (*add0n.com/pdf.html*, *http://nato.com/payload.php*, and *http://nato.com/upload.php*) are parsed to keywords records. The invocation of *app.launchURL(...)* is parsed to an API record.