Fear and Logging in the Internet of Things

Qi Wang, Wajih Ul Hassan, Adam Bates, Carl Gunter
University of Illinois at Urbana-Champaign
{qiwang11, whassan3, batesa, cgunter}@illinois.edu

***Abstract*—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 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. We prototype ProvThings for the Samsung SmartThings platform, and benchmark the efficacy of our approach against a corpus of 26 IoT attacks. Through the introduction of a selective code instrumentation optimization, we demonstrate in evaluation that ProvThings imposes just 5% overhead on physical IoT devices while enabling real time querying of system behaviors, and further consider how ProvThings can be leveraged to meet the needs of a variety of stakeholders in the IoT ecosystem.**

I. Introduction

The rapid expansion of the Internet of Things (IoT) is providing great benefits to our everyday lives. 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. 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 [23], Apple's HomeKit [14], and Google's Android Things [13] are just a few examples. IoT platforms offer appified software [31] 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 [61], [69], [67], [10]. In some cases, IoT attacks could have chilling safety consequences – burglars can now attack a smart door lock to break into homes [51], and arsonists may even attack a smart oven to cause a fire [42]. 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 [23], [29], [30], [20]. Some provided high-level event descriptions (e.g., *"Motion was detected by Iris Indoor Camera at 11:13 AM"*) [20], while others exposed verbose but obtuse low-level system logs [29]. 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 [41], 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 (§II) 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 [54], which can

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23282
www.ndss-symposium.org

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 [38], [47], [59]. 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.

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 (§IV). 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 (§V), and exhaustively evaluate the efficacy and performance of our prototype (§VI). 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 (§VII), 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.

II. Background

A. 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 [33]. Many of them, such as SmartThings and AWS IoT [19], 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.

[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

TABLE I: 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 [23]	cloud	hub	Y	181 [a]
Wink [30]	cloud	hub	N	N/A
Iris [20]	cloud	hub	N	N/A
Vera [29]	hub	hub	Y	236 [b]
HomeKit [14]	hub	hub	N	N/A
Android Things [13]	cloud	cloud	Y	12 [c]

As IoT is a sprawling and diverse ecosystem, in this work we focus on home automation platforms, which have the largest market share of IoT consumer products [33]. 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 I 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 [4]. Currently, the cloud-centric architecture is the most popular architecture [48], an example of which is shown in Figure 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 of a trigger-action programming paradigm [72].

Unfortunately, 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 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

[1]Different IoT platforms use different terms to refer the same concepts. For example, a physical smart device is termed *device* [25] in Samsung's SmartThings, while is termed *accessory* [18] in Apple's HomeKit.

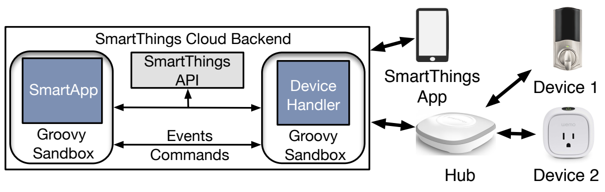


Fig. 1: SmartThings architecture overview.

```
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)
}
```

Fig. 2: An example SmartApp that monitors the events of a smart lock.

would therefore sound an alarm [44]. Diagnosing errors is also difficult in benign environments. An error in one rule may lead to unexpected behaviors [52], [60], 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.

Samsung SmartThings. Due to its maturity, in this 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 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 [28] 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 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. 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 events of a device (e.g., motion detected).

B. 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?"*. Data provenance supports a wide variety of applications such as network troubleshooting [36], [40], [73], forensic analysis of attack [58], [56], and secure auditing [77], [38]. It therefore stands to reason that data provenance would be an invaluable tool within IoT.

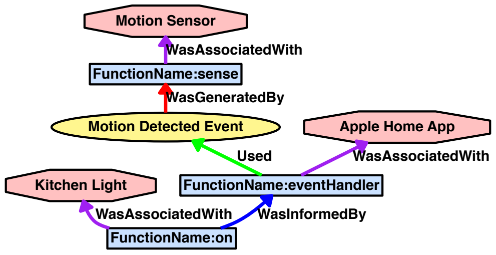


Fig. 3: An example provenance graph that describes why a kitchen light was turned on by Apple HomeKit.

Data provenance could allow us to understand the causal relations within a smart home. An example of an IoT provenance graph is shown in Figure 3 describing the circumstances under which a kitchen light was turned on by Apple HomeKit. The bottommost node in the graph represents a service[2] 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 [2] 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 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.

III. Threat Model & 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 [44], [54].
- **Device Vulnerability**: An attacker may gain remote access to a device through accessing an inadequately protected management interface [1], [61].

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

TABLE II: 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, ..

- **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 [49].

What our work does *not* consider is an attacker that can obtain root access to devices (e.g., Mirai attack [10]), but instead assumes device integrity. The assumption of device integrity has been used consistently in closely-related prior work [44], [45], [54]. 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 [6], 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., [34]). 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.

IV. 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 I. 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.

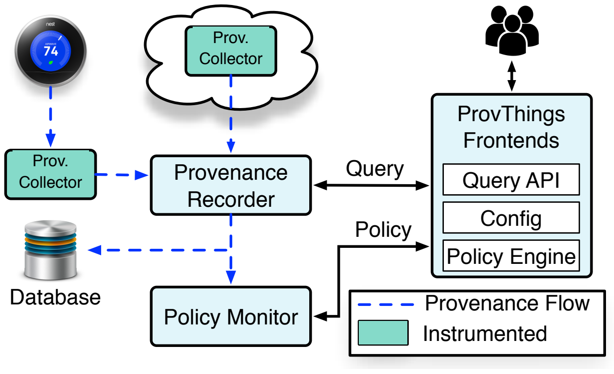


Fig. 4: The architecture of the ProvThings provenance management framework.

IoT Provenance Model. Our approach to addressing these requirements in ProvThings is to identify the common concepts present in different IoT platforms from §II and define a unified IoT provenance model based on the W3C PROV-DM [2]. 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 II. We map each concept to the PROV model and use a `subtype` property to further categorize concepts. For example, a smart 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 [24]. By default, we consider *device state*, *device event*, *device message* and *input* as sources. And we consider *device command* and *action* as sinks. In §VI, 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. 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 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 §V and discuss the implementations for two other platforms in §VIII. 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 [21]. 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 §VII. 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 minimality, 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

Algorithm 1: The Selective Code Instrumentation Algorithm.

```
Inputs : $ast \leftarrow$ Abstract Syntax Tree of a Program;
         $entries \leftarrow$ Program Entry Points;
         $sources \leftarrow$ Source Set;
         $sinks \leftarrow$ Sink Set;
Output: $instAst \leftarrow$ Instrumented ast
1  foreach $method \in ast.methodNodes$ do
2      if not IsReachableFromEntry (method, entries) then continue
3      if not $method.name \in sinks$ then continue
4      if $method.name \in entries$ then
5          AddInstrument(method) /* Insert code to create an Activity and create Used relations with arguments. */
6      foreach $branch \in method.branches$ do
7          foreach $stm \in branch.statements$ do
8              if stm is MethodCall and $stm.name \in sinks$ then
9                  AddInstrument(method) /* Insert code to create Activity, Used relation, and WasInformedBy relation with the top method in call stack */
10                 $varsUsed \leftarrow$ all variables in $stm$.arguments
11                 $sourceVars \leftarrow varsUsed \cap sources$
12                 if $sourceVars \neq \emptyset$ then
13                     $slice \leftarrow$ BackwardSlice($stm$, $sourceVars$)
14                     foreach $stm2 \in slice$ do
15                         AddInstrument($stm2$) /* Insert code to create Entities and WasDerivedFrom relations */
```

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 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* [78] 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 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 13). (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
```

Fig. 5: 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 5. 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 [16], 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
```

Fig. 6: 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 6, 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

TABLE III: 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 [45]	✓	✓	✗	✗	✗
ContextIoT [54]	✓	✗	✗	✓	✓
ProvThings	✓	✓	✓	✓	✓

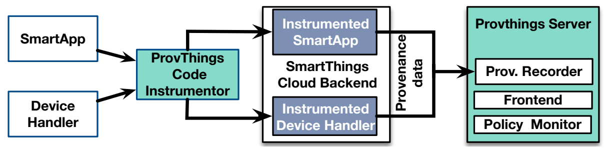


Fig. 7: An overview of the deployment of ProvThings on the SmartThings platform.

the sink function. Otherwise, the control code goes to the next statement. In §VII-C, 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 summarized in Table III. 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 §VII-C 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.

V. 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 7, 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 [47] 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 §IV 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 [15] at the semantic analysis pass of compilation. To implement Algorithm 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[3]. 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[4], we identified 85 device commands protected by 89 capabilities are supported by SmartThings.

As shown in Algorithm 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 §VIII. The other exception was assignment to global variables as they are shared 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 8, we show the instrumented version of the example SmartApp in Figure 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

[3]http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/
[4]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 }
```

Fig. 8: Instrumented version of the example SmartApp shown in Figure 2. The instrumented code is highlighted in grey background.

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 8 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 §IV, 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.

VI. 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 [26], 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 1) in order to only generate provenance records related to `sources` and `sinks`.

A. Effectiveness

To evaluate the completeness of ProvThings, we constructed SmartApps for a corpus of 26 possible attacks on IoT platforms through surveying relevant literature [44], [54], [61], [69]. 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 [44] and covers 22 out of 25 attacks used in the evaluation of [54].

To establish a ground truth for describing the complexity of each attack, two coders (authors of this paper) 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 V, 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 §IV. 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.

TABLE IV: 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 [44]	7	8	7	8	100%
Door Lock Pin Code Snooping [44]	23	27	23	27	100%
Disabling Vacation Mode [44]	19	17	19	17	100%
Fake Alarm [44]	14	13	14	13	100%
Creating seizures [67], [54]	173	168	173	168	100%
Surreptitious Surveillance [54]	34	31	34	31	100%
Spyware [5]	10	10	10	10	100%
Undesired unlocking [51], [54]	6	5	6	5	100%
BLE relay unlocking [51], [54]	7	5	7	5	100%
Lock Access Revocation [51], [54]	18	29	18	29	100%
No Auth Local Command [69]	7	5	7	5	100%
No Auth Remote Command [61]	7	5	7	5	100%
Repackaging [54]	15	15	15	15	100%
App Update [54]	6	5	6	5	100%
Drive-by Download [54]	14	11	14	11	100%
Remote Command [54]	13	13	13	13	100%
User Events [54]	12	13	12	13	100%
System Events [54]	29	31	29	31	100%
Abusing Permission [54]	9	8	9	8	100%
Shadow Payload [54]	28	31	28	31	100%
Side Channel [54]	61	59	61	59	100%
Remote Control [54]	14	14	14	14	100%
Adware [54]	19	16	19	16	100%
Ransomware [54]	29	25	29	25	100%
Specific weakness [54]	29	33	29	33	100%
IPC [54]	91	81	91	81	100%

TABLE V: 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

B. 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 V. ProvFull has larger instrumentation time and introduces more LoC as compared 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 (§V).

C. 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 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).

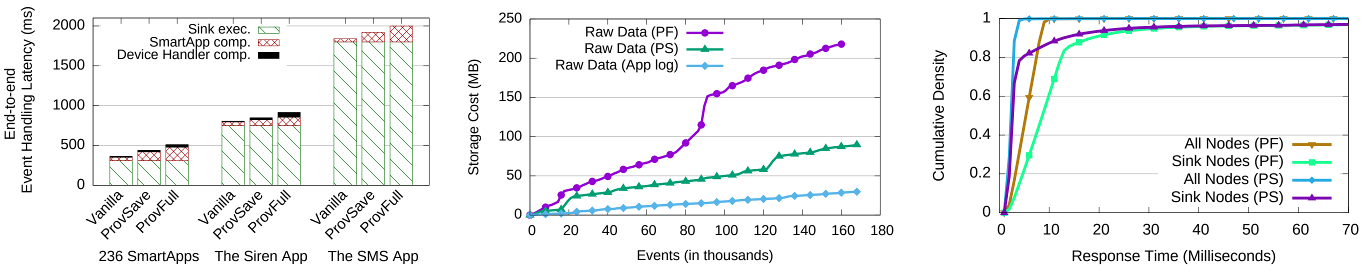
The SmartThings cloud IDE provides a simulator which can model the behavior of physical devices with virtual devices. In the experiment, we run our corpus of 236 SmartApps within the simulator. To automate the test, we build an automatic testing framework using Selenium [22] which automatically install 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, we generate both `lock` and `unlock` event to trigger the `eventHandler`. Our results are shown in Figure 9a. 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 [12] if the gun case is moved, and a SmartApp that sends an SMS to the user's phone when power consumption exceeds a threshold.[5] 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.

Storage Overhead. We determine the storage costs of provenance collection by measuring log growth during our runtime performance tests, shown in Figure 9b. 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 [3], which would translate to just 260 KB storage cost for ProvSave. We thus conclude that ProvThings imposes negligible storage costs.

D. 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 §IV. For each node, we request the ancestry of it to produce a provenance graph. The query performance is shown in Figure 9c. For graphs with 2 million nodes generated by ProvSave, the average query time for all

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



(a) End-to-end event handling latency overhead **(b)** Provenance storage growth **(c)** Provenance query performance

Fig. 9: Runtime Overheads for ProvThings; PF: ProvFull, PS: ProvSave

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.

VII. 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.

A. 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 B 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 10. 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, especially when the value used by the dynamic invocation was from an untrusted source. Based on the provenance graph shown in Figure 10, she creates a policy as shown in Figure 11. 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 [11], misbehaving SmartApps [8] and unmatched Device Handlers [9]. 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 issue [11]. With ProvThings, Marc could first query all the `off` activities of the kitchen light that were created during the

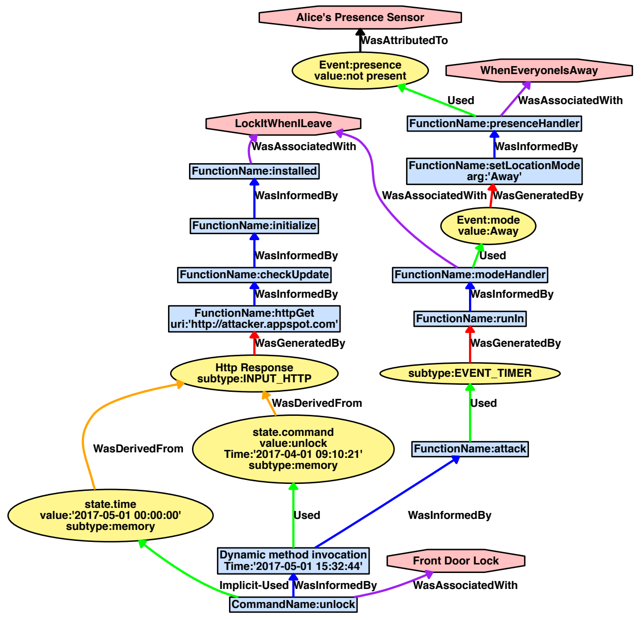


Fig. 10: The provenance of an unintended unlock event for a front door. The app `LockItWhenILeave` visited a malicious domain to retrieve a command, then waited until after a specified time when the mode was away to execute the command.

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

Fig. 11: A policy to detect vulnerable dynamic method invocations use values from an HTTP input.

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 [7], 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.

B. 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

```
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
```

Fig. 13: A policy to detect unintended information flows.

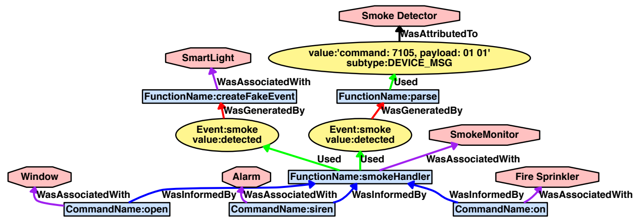


Fig. 14: A provenance graph shows the provenance of a real smoke event and a fake smoke event.

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 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 [44] 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 12 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 13. 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.

C. 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 [44], in this case, we consider a user installed a benign app (`SmokeMonitor`) which monitors the events of

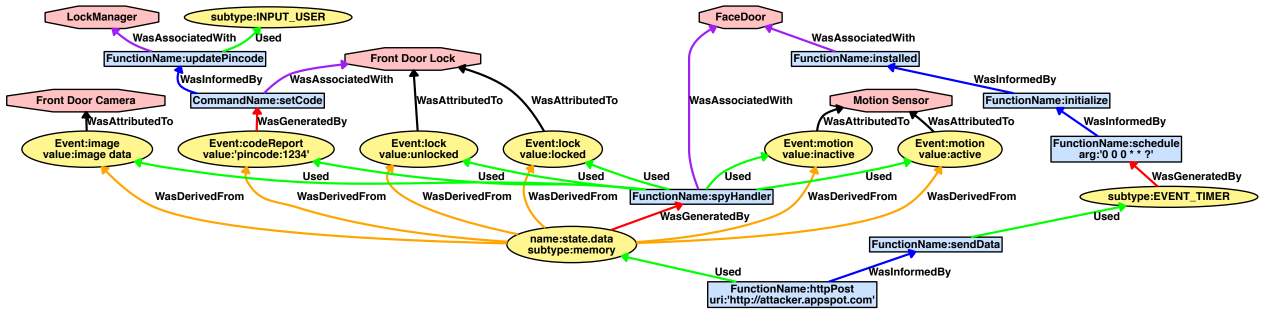
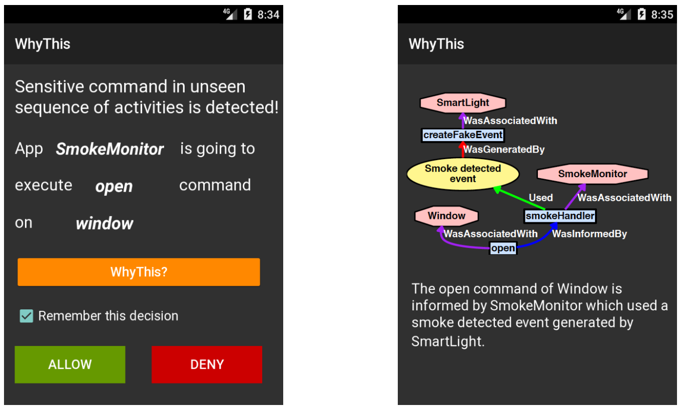


Fig. 12: A provenance graph shows how some sensitive information (for example the lock pin code) was leaked. The `spyHander` function collected sensitive information and a scheduler triggered the `sendData` function to send the data to the attacker.


8:34

WhyThis

Sensitive command in unseen sequence of activities is detected!

App ***SmokeMonitor*** is going to execute ***open*** command on ***window***

WhyThis?

Remember this decision

ALLOW DENY

8:35

WhyThis

The open command of Window is informed by SmokeMonitor which used a smoke detected event generated by SmartLight.

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

Fig. 15: 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
```

Fig. 16: WhyThis procedurally generates a policy to deny fake smoke events from SmartLight.

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 14. 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 15, 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 15a. The user can click the *WhyThis?* button to see a simplified provenance graph and a paragraph description before making a decision (Figure 15b). 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 in Figure 16. 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 [68] to meet their usability requirements for typical consumers.

D. *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 [27]. 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 [38] 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.

VIII. 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 [17]. 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 I. For Vera [29], we could perform source code instrumentation to its Lua-based apps. For Android Things [13], 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.

IX. Related work

IoT Security. A lot of vulnerabilities have been identified in IoT devices [62], [61], [69], [51], [67], [10] and protocols [46], [6]. Fernandes et al. [44] 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. [70] 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. [74] 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 [45] is a system that enforces flow policies for IoT apps to protect sensitive data. ContextIoT [54] is a context-based permission system for IoT platforms which collects context information to identify sensitive actions. As compared in §IV, 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 [65], [55] have been proposed for IoT forensics. Oriwoh et al. [63] 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. [76] 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 [38], [56], [58], [66], [57], network debugging, auditing [50] and troubleshooting [36], [73], [40], and intrusion detection and access control [37], [64]. Similarly, provenance-based solutions are proposed for android to provide attack reconstruction [43], [35], [75], debugging and diagnosing device disorders [53]. 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 [39], [32], [71] 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 paper. 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.

X. 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, enabling 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.

Acknowledgements

We thank our shepherd Aziz Mohaisen and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF CNS grants 15-13939, 13-30491, and 16-57534. The views expressed are those of the authors only.

References

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

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

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

[4] "3 Types of Software Architecture for Internet of Things Devices," https://goo.gl/u9NTXS, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] 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.

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

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

[34] 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.

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

[36] 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.

[37] 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.

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

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

[40] 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.

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

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

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

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

[45] 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.

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

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

[48] 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.

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

[50] 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.

[51] 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.

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

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

[54] 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.

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

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

[57] ——, "LogGC: garbage collecting audit log," in *CCS*, 2013.

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

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

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

[61] 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.

[62] 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.

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

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

[65] 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.

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

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

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

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

[70] 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.

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

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

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

[74] 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.

[75] 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.

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

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

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

Appendix

A. 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.

```
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()
}
```

B. 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.

```
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. 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.

```
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;
}
```