IOTA: A Framework for Analyzing System-Level Security of IoTs

Zheng Fang*, Hao Fu*, Tianbo Gu*, Pengfei Hu†, Jinyue Song* Trent Jaeger‡, and Prasant Mohapatra*
* Department of Computer Science, University of California, Davis
† School of Computer Science and Technology, Shandong University
‡ Department of Computer Science and Engineering, Pennsylvania State University
Email: {zkfang,haofu,tbgu,jysong,pmohapatra}@ucdavis.edu, phu@sdu.edu.cn, tjaeger@cse.psu.edu

***Abstract*—Most IoT systems involve IoT devices, communication protocols, remote cloud, IoT applications, mobile apps, and the physical environment. However, existing IoT security analyses only focus on a subset of all the essential components, such as device firmware, and ignore IoT systems' interactive nature, resulting in limited attack detection capabilities. In this work, we propose IOTA, a logic programming-based framework to perform system-level security analysis for IoT systems. IOTA generates attack graphs for IoT systems, showing all of the system resources that can be compromised and enumerating potential attack traces. In building IOTA, we design novel techniques to scan IoT systems for individual vulnerabilities and further create generic exploit models for IoT vulnerabilities. We also identify and model physical dependencies between different devices as they are unique to IoT systems and are employed by adversaries to launch complicated attacks. In addition, we utilize NLP techniques to extract IoT app semantics based on app descriptions. To evaluate vulnerabilities' system-wide impact, we propose two metrics based on the attack graph, which provide guidance on fortifying IoT systems. Evaluation on 127 IoT CVEs (Common Vulnerabilities and Exposures) shows that IOTA's exploit modeling module achieves over 80% accuracy in predicting vulnerabilities' preconditions and effects. We apply IOTA to 37 synthetic smart home IoT systems based on real-world IoT apps and devices. Experimental results show that our framework is effective and highly efficient. Among 27 shortest attack traces revealed by the attack graphs, 62.8% are not anticipated by the system administrator. It only takes 1.2 seconds to generate and analyze the attack graph for an IoT system consisting of 50 devices.**

***Index Terms*—Internet of Things (IoT), Security and Privacy, Attack Graph**

I. INTRODUCTION

The last decade witnessed the rapid development and wide deployment of IoT systems. According to [3], the total global worth of IoT technology could be as much as 6.2 trillion US dollars by 2025. Popular commodity IoT platforms, such as Samsung SmartThings [41], Apple Homekit [8], and Google Nest [23], etc., share similar architecture: low power end devices running customized firmware, short-range, wireless communication protocols, a centralized decision-maker, IoT applications using trigger-action paradigms, and companion mobile apps. IoT components interact with each other in sophisticated ways. For example, devices' functionality depends on secure and reliable communication with the controller, and devices can be dependent on each other due to IoT applications or physical channels. The distributed but interactive components pose tremendous challenges to IoT system security verification and analysis [12], [20], [22], [32], [47].

Existing research on IoT security only focuses on a single or a subset of the IoT components. For instance, [16], [18], [49] analyze IoT device firmware, [39] investigates IoT wireless protocols, and [12], [32], [44] sanitize IoT applications. However, for interconnected systems, hardening individual components cannot guarantee security because there are multiple paths to compromise system resources. For example, attackers can unlock a smart doorlock by exploiting vulnerabilities on the lock [1], but they may also compromise an indoor camera [2] and use it to inject voice, triggering a smart speaker to launch the door-open command [5]. In this paper, we try to address the following research problem — How to verify IoT systems security and uncover threats in a systematic way?

Attack graphs [7], [35], [40] provide us an elegant approach to the problem by enumerating all of the paths to potential *attack goals*, i.e., system resources which can be compromised by the attacker. There are two types of attack graphs: *state-based attack graph* [38], [40] and *exploit-dependency attack graph* [7], [35]. State-based attack graphs utilize model checking as the reasoning engine. But they suffer scalability issues in that the size of the graph grows exponentially with the number of system state variables (The number of system state variables is a linear function of the system device count). In comparison, it takes polynomial time to construct exploit-dependency attack graphs, and the generated attack graph size is a quadratic function of the system device count [35].

However, existing exploit-dependency attack graph frameworks cannot be readily applied to IoT systems due to multiple design limitations. First, existing exploit-dependency attack graphs were designed for conventional computer networks and do not model essential IoT components such as IoT apps and devices' physical dependencies. Second, many IoT devices communicate using low-power protocols such as Zigbee or ZWave, which most of the existing vulnerability scanners cannot scan. For example, all of the vulnerability scanners listed on [46] only support IP-based devices. Moreover, existing frameworks do not model exploits on low-power, short-range protocols which are ubiquitous in IoT systems. Finally, there are no quantitative criteria for administrators to harden the system in such a way that vulnerabilities with the largest

impacts get patched first. As today's IoT systems may contain hundreds of vulnerabilities, it is necessary to patch vulnerabilities efficiently.

Goals. In this paper, our goal is to build a system-level security analysis framework for IoTs which, given the IoT system configurations (i.e., device, network information, and the IoT apps installed), (a) constructs exploit-dependency attack graphs to uncover resources that can be compromised and reveal potential attack traces; and, (b) computes a suite of metrics to interpret the generated attack graph and provide recommendations for system hardening.

As exploits and devices' dependencies are the key building blocks of attack graphs, to achieve (a), we extract exploit models and device dependencies from IoT system configurations and represent them as Prolog clauses [36]. More specifically, IOTA scans IoT system configurations for individual vulnerabilities and builds *exploit models* (consisting of precondition and effect) based on scanned CVEs. We identify three types of device dependencies: *app-based dependency*, *indirect physical dependency*, and *direct physical dependency*. The app-based dependencies are specified by IoT app semantics (i.e., trigger-action rules). Since IoT apps' source code can be unavailable in some platforms, such as IFTTT [27], we utilize natural language processing (NLP) techniques to extract app semantics from app descriptions. The direct and indirect physical dependencies are universal in IoT systems and thus are hard-coded as Prolog rules. Finally, Prolog clauses are sent to MulVAL [36] to generate attack graphs.

With regards to (b), we propose two novel metrics: *shortest attack trace* to an attack goal, and *blast radius* of a vulnerability. The shortest attack trace to an attack goal node provides the lower bound of the attack complexity in terms of the number of exploits to launch. A vulnerability's blast radius tells us the potential capabilities the attacker can get on the system by exploiting *only* that vulnerability. In addition, the concept of *attack evidence* (defined to help us compute blast radius) can also be used to compute the *minimal set of vulnerabilities to patch to thwart an attack goal* [40]. These metrics help administrators interpret the attack graphs, sort the vulnerabilities based on their impacts on the system, and make informed choices about system hardening.

To evaluate IOTA, we generate 37 synthetic smart home IoT systems based on 532 real-world IoT apps and a list of 59 smart home devices of 26 types. We scan the CVE database since 2010 and find 127 IoT CVEs on those 59 smart home devices. Our vulnerability analysis module achieves 80.56% accuracy for exploit precondition identification and 88.19% accuracy for exploit effect. We manually check 27 shortest attack traces whose depths are at least 9 and find out 62.8% of them are beyond anticipation. In particular, the graph analyzer module reveals a shortest trace of depth 18 for an IoT system consisting of only 7 devices, implying that attack traces can be very deep for even a small IoT system. The case study illustrates the effectiveness of using the shortest attack trace and blast radius to estimate attack complexity and their impacts on the system. IOTA is highly scalable. In practice, it only takes around 1.2s and 120MB memory to evaluate IoT systems of 50 devices.

In summary, we make the following contributions:

- We introduce IOTA, a novel framework to conduct automatic, system-level IoT system security analysis and generate attack graphs showing all potential attack traces.
- We design formal models for IoT exploits and implement automatic translation from system configuration and vulnerability information to Prolog clauses.
- We propose two metrics to quantitatively evaluate the attack complexity (shortest attack trace) and vulnerability's system-level impacts (blast radius).
- We evaluate the efficiency and effectiveness of IOTA by applying it to 37 synthetic IoT systems of different sizes ranging from 4 to 50 and verify that our framework is both effective and highly efficient.

II. THREAT MODEL

In this work, we consider individual attackers whose goal is to break into the system. They can be physically adjacent to the IoT system, enabling them to be within the radio range of the wireless local area networks, such as WiFi or Zigbee networks. Besides, the attacker can physically access outside, unprotected IoT devices, such as a doorbell or outdoor surveillance cameras. We also assume the attacker is able to extract IoT app semantics because he can install sniffers and infer event type from the sniffed packets [24]. We treat the remote IoT cloud as trustworthy and do not consider the compromise of the cloud itself. However, if there exist vulnerabilities on the companion mobile app, the attackers can spoof commands to the remote cloud. Below we discuss the major threats to each of the IoT components in detail.

A. Device

We use the term "IoT devices" to refer to both end devices and infrastructure devices such as routers and gateways. Most of the device vulnerabilities are rooted in the firmware [16], [17], [26]. However, some vulnerabilities are found in the device's physical components [42], [43]. Once a device is compromised, it can be used to attack other components of the IoT system in three different ways. First, if the attacker gets root privilege on a device such as a router, he can send spoofed commands to other devices on the same network. Second, the attacker can utilize the compromised device to inject cyber events, such as spoofing a motion event. Moreover, the attacker can take advantage of the devices' physical dependencies to compromise other devices.

B. Network

IoT systems utilize short-range, low-power protocols to communicate with the end devices, which allows adjacent attackers to sniff the wireless traffic. These end devices are first connected to a gateway (sometimes called base station, bridge, or hub) in order to communicate with the internet. Since generally there is no firewall or MAC address filtering in most smart home networks, if the attackers gain access to

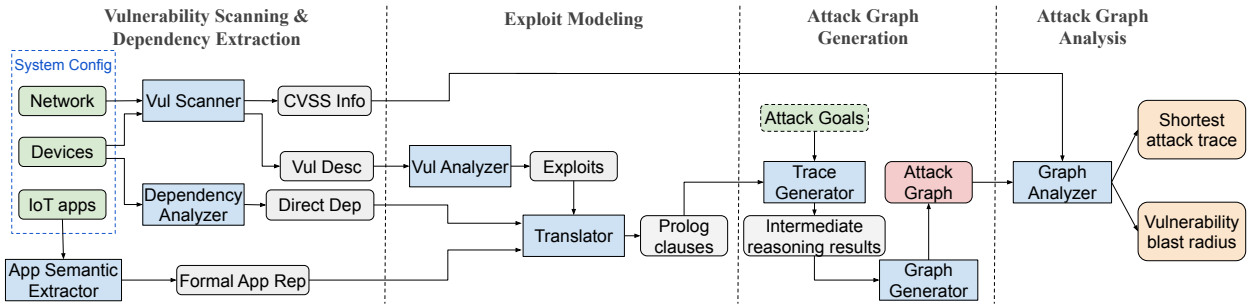


Fig. 1: IOTA pipeline. The blue boxes are IOTA modules. The green, red, and gray boxes represent input, output, and intermediate results, respectively. The *Attack Goals* can be set by the system administrator and is optional.

the network, they can send spoofed packets to other devices on the same network. To make things worse, many IoT devices, such as routers, cameras, or thermostats, expose unprotected network services to the home network, making it possible for the attackers to compromise these devices after they join the home network.

C. IoT application

IoT applications are designed using the *if-this-then-that* paradigm, where *this* represents IoT event(s) and *that* represents some device actions. IoT apps introduce dependencies among devices, which expose new attack surfaces to the adversary. Consider "If smoke is detected, sound the alarm and open the window." as an example. To open the victim's window, the attacker does not have to attack the window opener directly; instead, he can just compromise the smoke detector, and the IoT app will do the rest of the attack for him. Even though the attacker must know the app has been installed before exploiting it, people have shown this can be done via wireless sniffing [24], [48].

D. Physical channel

One of the unique features of IoT systems compared with other networked systems is that IoT devices can interact with each other via the *physical channels*. There is a distinction between the IoT physical channels and the physical layer of the computer networks: The former is shared physical environments, such as air, temperature, and humidity, whereas the latter is electromagnetic signals transmitting raw bitstreams. While IoT app-based device dependencies will only exist if the app is installed by the user, physical device dependencies *always* exist in an IoT system as long as the relevant devices are installed. An attacker can utilize various physical dependencies to launch attacks. For instance, he can first compromise the indoor camera, e.g., Nest Cam IQ Indoor, and use it to inject human voice commands. The smart speaker will receive the voice and issue the corresponding command to the actuator.

III. System Overview

Figure 1 illustrates the pipeline of IOTA, which consists of four stages. **Vulnerability Scanning and Dependency Extraction** stage scans the devices and network protocols for vulnerabilities and extracts IoT app semantics. It also extracts direct device dependencies from the system configuration file. **Exploit Modeling** stage maps vulnerabilities to exploits based on the vulnerability description and Common Vulnerability Scoring System (CVSS) [14] scores such as Attack Vector and Confidentiality, Integrity, Availability (CIA) triad. Exploits, direct dependencies, and app-based device dependencies are then translated to Prolog clauses. **Attack Graph Generation** stage reads attack goals (optional) and the translated Prolog clauses and generates IoT attack graphs. If the user does not provide attack goals, we enumerate all system resources (i.e., privileges on devices or tamper of the physical features) as potential attack goals. Then we modify the intermediate reasoning results and send them to MulVAL [36] to generate attack graphs. **Attack Graph Analysis** stage takes the generated attack graph as input and computes the following metrics: the shortest attack traces to each attack goal node and the blast radius of each vulnerability.

IV. IOTA Design

In this section, we explain the design of the IOTA modules as shown in Figure 1. The implementation details are explained in Section V.

A. Vulnerability Scanner

To the best of our knowledge, there are no existing vulnerability scanners readily available for low power communication protocols such as Zigbee or Bluetooth Low Energy (BLE). Therefore, we design a vulnerability scanning approach based on CVE database searching. Our approach is practical because of some device vendors' ignorance of vulnerability report [33], [37] and the slow firmware upgrade rate [18].

Given the IoT devices installed and the communication protocols used, the Vulnerability Scanner module searches the CVE database [13] for vulnerabilities. We fetch the CVE JSON feeds since 2010 from the National Vulnerability Database (NVD) [31] and parse the JSON files to get information relevant to our exploit modeling, including impact score, exploitability score, exploit range, exploit result (CIA triad), and the vulnerability description. The parsing results are stored in a local MySQL database. In total, there are 121,210 CVEs from 2010 to April 2021. After discarding CVEs without CVSS information, our database contains 113,180 records. For each device instance listed in the system configuration file, we

TABLE I: IoT device direct dependencies and examples.

Direct Dependency	Example
Electrical	Outlet → AC; Switch → Light bulb
Mechanical	Door lock → Door opener
Utility	Water valve → Sprinkler; Gas valve → Stove

query the database for the device name using full-text search in boolean mode to make sure it only returns CVE records when all of the query keywords appear in the CVE description.

The scanned vulnerability on each device is then translated to Prolog facts. For example, the following fact shown in Listing 1 means vulnerability `CVE-2020-8864` exists on `dLinkRouter`.

```
vulExists(dLinkRouter, 'CVE-2020-8864').
```

Listing 1: Prolog fact for a CVE found on a device.

B. Dependency Analyzer

The Dependency Analyzer module models how IoT devices interact with each other via physical channels. We identify and define two types of physical dependencies: direct dependency and indirect dependency. Two devices are *directly dependent* if they are both actuators. There are three types of direct dependencies as listed in Table I. The most common direct dependency is **electrical dependency**, such as the one between smart outlet and air conditioner. The second type is **mechanical dependency**. For example, the door opener cannot open the door if the door lock is locked. We define the third type as **utility dependency**. For example, gas valve and smart stove are dependent via gas. Even though direct dependencies can have a huge impact on IoT system security, they are overlooked by existing IoT security analysis frameworks.

Two devices are *indirectly dependent* if one of them is an actuator and the other is a sensor. We consider and model six physical channels: temperature, humidity, illuminance, voice, smoke, and water. We include "voice" as a physical channel because many devices like cameras and TVs can play human voice in the smart home, and some devices can recognize human voice and execute the corresponding instructions.

Direct and indirect physical dependencies are hard-coded as Prolog rules because they are universal in all IoT systems, regardless of the installation of certain IoT apps. During the execution of a Prolog program, a certain dependency rule will be activated only when the corresponding device is installed. For example, if AC is on, then the room temperature will be low. But if there is no temperature sensor installed, the predicate of sensor reporting low temperature will not hold true. Listing 2 and Listing 3 are example of Prolog rules for indirect and direct dependencies, respectively.

```
high(temperature) :-
    on(Heater),
    heater(Heater).

reportsHigh(TemperatureSensor, temperature) :-
    high(temperature),
    temperatureSensor(TemperatureSensor).
```

Listing 2: Indirect physical dependency.

```
off(Device) :-
    plugInto(Device, Outlet),
    outlet(Outlet),
    off(Outlet).
```

Listing 3: Direct physical dependency.

C. App Semantic Extractor

The App Semantic Extractor module extracts semantic information from IoT app descriptions using NLP techniques. Compared with program analysis, analyzing IoT app descriptions in NLP is more applicable in that app descriptions are publicly available while IoT apps' code may be proprietary on some platforms. In smart home platforms, developers write a short description to explain the functionality of their IoT apps to smart home users. Typically, these app descriptions are written in "If this, then that" form, which makes it suitable for NLP techniques.

We use Stanford CoreNLP framework [29] and Natural Language Toolkit (NLTK) [10] for app description analysis. Given an app description, we use CoreNLP parser to construct the constituency parse tree and split the sentence into the conditional clause and the main clause based on tree node labeled `SBAR` (subordinate clause). We do a breadth-first search on the parse tree to find the tree node with label `SBAR`, which is the root of the conditional clause. Then the conditional clause is obtained by concatenating the leaf nodes of the subtree whose root node has label `SBAR`. We construct the main clause by removing the conditional clause string from the whole sentence.

Because the conditional clause and the main clause may contain multiple conditions or actions, we further split each clause into simple sentences based on tree node labeled `CC` (coordinating conjunction). The coordinating conjunction represents either logic AND or logic OR relationship between the two simple sentences. For example, the split of SmartApp *Hall Light: Welcome Home*'s description "Turn on hall light if someone comes home and the door opens." is shown in Listing 4. The conditional clause is split into two simple sentences with logic AND relationship. Since the main clause contains just one simple sentence, the relationship is set to `'NONE'`.

```
conditional:  ('AND', ['someone comes home', 'the
    door opens'])
main:  ('NONE', ['Turn on the hall light'])
```

Listing 4: Splitting clauses into simple sentences for the SmartApp *Hall Light: Welcome Home*.

After splitting each clause into simple sentences, we extract noun and verb phrases from each simple sentence and match them to IoT device names and device action using Word2Vec similarity. We use a regular expression chunker to extract noun phrases and verb phrases. The regular expression patterns we use for chunking and the extracted phrases for the SmartApp description are shown in Listing 5 and Listing 6, respectively.

```
NP: {<DT>?<JJ>*<NN.*>+}
VP: {<VB.*><IN|RP>?}
```

Listing 5: Regular expression patterns.

```
conditional clause: [(['someone'], ['comes']), (['
    the door'], ['opens'])]
main clause: [(['the hall light'], ['Turn on'])]
```

Listing 6: Noun and verb phrases extracted for the SmartApp *Hall Light: Welcome Home*'s description.

Finally, we use Word2Vec model [30] to match the extracted noun phrases and verb phrases with our pre-defined list of devices and device actions. Since Word2Vec only computes similarities between individual words, we compare each word in a noun phrase against each word in a device name. The app semantic extraction result is represented as a Python tuple shown in Listing 7. This internal representation is used together with app configuration information in the Translator module to generate Prolog rules.

```
('AND', ['motion sensor', 'door contact sensor'], ['
    motion', 'open'], 'NONE', ['bulb'], ['on'])
```

Listing 7: Internal representation of the IoT app semantic.

D. Vulnerability Analyzer

The Vulnerability Analyzer module maps vulnerabilities to exploit models. Exploit modeling is essential for attack graph construction because attack traces are composed of individual exploits. To the best of our knowledge, our work is the first to attempt to *automatically* generate exploit models based on CVEs' natural language description and CVSS scores. Though MulVAL [36] formally represents exploits as Prolog rules, it only considers privilege-escalation in computer networks. Our exploit models are designed for generic IoT systems and consist of exploit precondition and effect. A *precondition* is the privilege the attacker should have in order to launch an exploit. An *effect* is the direct result of an exploit.

Precondition. For IoT systems, we define five types of preconditions listed in Table II. Because IoT systems typically involve low-power, short-range, wireless protocols such as Wifi or ZigBee, the physically or logically adjacent precondition should be defined for each network type specifically, such as `Wifi adjacent logically`, `Zigbee adjacent physically`, etc. We cannot just use the "attack vector" value as the precondition of each CVE in the NVD database because that field can be ambiguous or sometimes incorrect: According to [19], it assigns "network" as the precondition whenever there is a lack of information to decide the exploit range. Besides, the value does not differentiate "physically adjacent" and "logically adjacent".

We predict the exploit precondition based on protocol type, CVE description, and the CVSS attack vector. If an exploit's attack vector is `local` or `physical`, we keep its value. If the attack vector is `adjacent`, we check its CVE description. If the description contains keywords such as "sniff", "decrypt" or their synonyms, we will assign the precondition as `adjacent physically`, otherwise `adjacent logically`. If the original attack vector is `network`, we will first check the protocol. If the protocol is a low-power protocol, then we invoke the approach of determining `adjacent`; otherwise, we set the precondition to `network`.

TABLE II: Types of exploit preconditions on IoT devices.

Precondition	Explanation
Network	An attacker can exploit the vulnerability from the internet. There's no prior privilege required on the IoT system.
Adjacent physically	The attacker needs to be within the radio range of a wireless network, but he/she does not need to be on the network.
Adjacent logically	The attacker should be both within the radio range and on the wireless network, in order to launch the exploit.
Local	The exploit requires access to the device with at least user privilege, such as establishing Telnet or SSH connection to the device.
Physical	The attacker needs physical access to the vulnerable device to commit exploits.

TABLE III: Types of exploit effects on IoT devices.

Effect	Explanation
Root	Attackers have root privilege on a device, which can be used to send spoofed commands to other IoT devices on the same network.
Device control	The attacker can run any command the device supports, and sniff and inject any device event. But the device cannot be used to send spoofed messages to other devices.
Command injection	The attacker can inject any commands to the device, but does not have access to the IoT events on that device.
Event access	The attacker is able to sniff and spoof events on an IoT device, but he/she cannot inject commands to that device.
Wifi access	The attacker obtains the Wifi credentials and is able to join the Wifi network.
DoS	The IoT device becomes denial-of-service.

Effect. From an attacker's perspective, exploit effects include gaining privileges on IoT devices, accessing wireless traffic, or making devices denial-of-service. We categorize exploit effects into six types as listed in Table III. If attackers get `root` privilege on a device, they can use it to attack other devices by sniffing or spoofing wireless traffic. The `device control` privilege implies both `command injection` and `event access` privilege, but not the capability of attacking other devices on the same system.

For each vulnerability, we decide its exploit effect based on the CVE description and confidentiality, integrity, and availability (CIA) subscores of CVSS. We first seek to extract the effect from the CVE description by matching the keywords for each effect type. If the description does not contain any keywords, we try to identify the *exploit mechanism* defined in [15] and communication protocol from the description using the same keyword matching approach. In combination with the CVSS' CIA subscores, we can infer the exploit effect. For example, suppose the exploit mechanism is buffer overflow. Then we check the CIA subscores to set the effect to `denial of service` (if only the availability score is greater than the

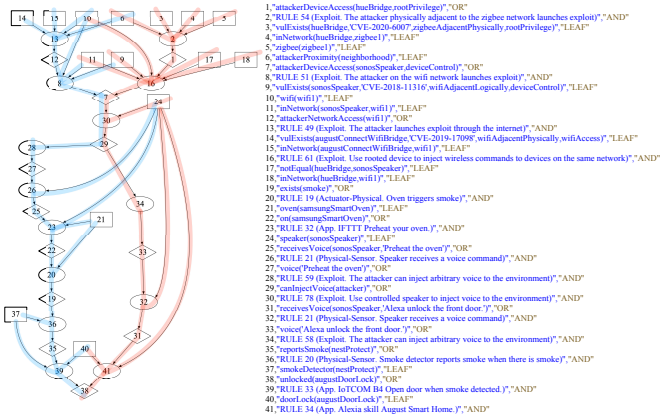


Fig. 2: IoT attack graph example. The meaning and type of each node is shown on the right.

threshold), or `root privilege` (if confidentiality, integrity, and availability are all greater than the threshold).

The exploit models are also translated to Prolog facts. For example, the `vulProperty` fact in Listing 8 is the exploit model for `CVE-2020-8864`. The precondition is the attacker being on the same Wifi network as `dLinkRouter`; the effect is that the attacker gets root privilege on this device.

```
vulExists(dLinkRouter, 'CVE-2020-8864').
vulProperty('CVE-2020-8864', wifiAdjacentLogically,
    rootPrivilege).
```

Listing 8: Prolog fact for an exploit model.

E. Attack Graph Generator

The Attack Graph Generator module takes the Prolog rule and fact file as input and verifies whether the attack goals (either provided by the administrator or automatically generated by IOTA) can be achieved. If a goal can be achieved, it will generate the attack graph showing all the attack traces; otherwise, the IoT system is protected from that attack goal. When the administrator knows his security objectives, he can set the attack goal by taking the logic NOT of the objectives. For example, if the objective is to protect the camera from being rooted, then the attack goal is the attacker's gaining root privilege on the camera. When there are no security objectives specified, we enumerate all the potential privileges attackers may get on all the devices as attack goals.

Figure 2 shows a small attack graph, where node 38 represents the attack goal — to unlock the doorlock. The meaning of each node is annotated on the right of the figure. In total, there are four *attack traces* (formally defined in Section IV-F) in the attack graph, and two of them are highlighted in red and blue. The attacker can reach node 7 (i.e., controlling Sonos speaker) by exploiting Hue bridge or August Wifi bridge. And from node 7, there are two ways to get to node 38: via the Alexa skill [9] (node 41) or by starting the oven to trigger smoke and using IoT app "IoTCOM B4" [6] (node 39).

Essentially, there are three kinds of nodes. The rectangle nodes represent *primitive facts* about the system state or the attacker state that are true before the exploit happens. The ellipse nodes represent Prolog *rules*, such as exploits or apps' execution. The diamond nodes stand for *derivation*, viz., new states about the system or the attacker after launching an exploit or executing an app. A derivation node can also be a precondition of another rule node. The logic meaning of each node is also annotated on the right of the figure. A detailed explanation of attack graph structure can be found in [35].

Algorithm 1: Shortest Attack Trace Algorithm

Input: (1) Attack graph G, (2) Attack goal node n
Output: Shortest attack trace to n on G

Algorithm `shortest_trace`(G, n)
  res_node = TraceNode(n)
  if n is leaf node **then**
    return $(0, res\_node)$
  /* If current node is OR node, take the minimum of the parent nodes */
  if n is OR node **then**
    Let l be the list of parent nodes of n
    $min\_depth = \infty$
    for each node m in l **do**
      $(cur\_len, cur\_pred) =$ `shortest_trace`(G, m)
      if $min\_depth > cur\_len$ **then**
        Set $res\_node.preds$ to m
        Update min_depth
    return $(min\_depth + 1, res\_node)$
  /* If current node is AND node, take the maximum of the parent nodes */
  if n is AND node **then**
    Let l be the list of parent nodes of n
    $min\_depth = -\infty$
    Set $res\_node.preds$ to l
    for each node m in l **do**
      $(cur\_len, cur\_pred) =$ `shortest_trace`(G, m)
      if $min\_depth < cur\_len$ **then**
        Update min_depth
    return $(min\_depth + 1, res\_node)$

F. Attack Graph Analyzer

Because the generated attack graph can be gigantic, containing thousands of nodes, it is impractical to visualize the graph. Therefore, we propose two metrics to extract critical attack traces and quantify the impact of vulnerabilities.

Shortest Attack Trace. Among all of the attack traces to a specific attack goal, the *shortest attack trace* takes the minimum number of exploits and provides a lower bound of the attack complexity to that goal node. For instance, the shortest attack trace to the goal node (node 5) in Figure 2 is highlighted in red whose depth is 12. Below we formally define the shortest attack trace and relevant concepts.

Definition. (Attack Trace) Given an attack graph G, an attack trace to a derivation node n is a subgraph G' satisfying the

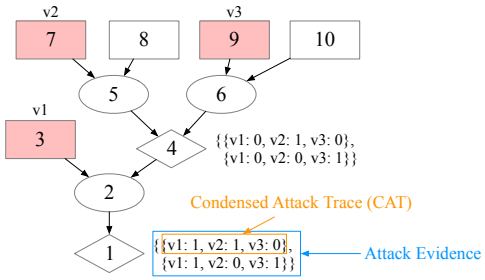


Fig. 3: Example attack graph and the corresponding attack evidence for node 1 and node 4. Node 3, 7, and 9 are primitive fact nodes describing different vulnerabilities represented as $v1$, $v2$, and $v3$.

following conditions: (1) Any OR node of G' has only one incoming edge; (2) Any AND nodes of G' has incoming edges from all its parent nodes; (3) All the source nodes of G' are primitive fact nodes; and (4) The sink node of G' is node n.

Definition. (Depth of an Attack Trace) The depth of an attack trace is the *longest path* from any primitive fact node to the sink node of the attack trace.

Definition. (Shortest Attack Trace) For a given attack graph and a derivation node n, the shortest attack trace is the attack trace to n with the smallest depth.

We cannot apply Dijkstra's algorithm to the shortest attack trace problem because our definition of shortest attack trace is different from the shortest path in graph theory: (1) There can be multiple source nodes; (2) The attack trace is a subgraph, not a path. Hence, we design a recursive algorithm, i.e., Algorithm 1, to compute the shortest attack trace to a specified attack goal node. The depth of a leaf node is defined as 0.

Blast Radius. The *blast radius* measures each vulnerability's impacts on the IoT system and can be used for system hardening. For example, if vulnerability A's blast radius is a superset of that of vulnerability B, we conclude that A's impact is bigger than B's, and therefore we should fix A first.

Definition. (Blast Radius (BR)) Given an attack graph, the blast radius of vulnerability v is the set of all of the privileges (represented as derivation nodes) the attacker gets after exploiting *only* v.

As there can be more than one trace to a certain node, and a vulnerability can be used in multiple attacks, we must keep track of vulnerabilities involved for each trace to a certain node in the attack graph. We come up with the following concepts to help us compute the blast radius of each vulnerability.

Definition. (Condensed Attack Trace (CAT)) Given an attack graph G, the condensed attack trace of a node n is the map from all of the vulnerabilities on G to 0 (when the vulnerability is not used) or 1 (when used) along some attack trace to n.

Definition. (Attack Evidence) The attack evidence of a node n is the set of its condensed attack traces.

Algorithm 2: Attack Evidence Merge — OR

Input: Attack Evidence of two nodes: a, b
Output: Attack Evidence of the child node c, an OR node

Algorithm `merge_ae_OR`(a, b)
  Let $merged\_ae$ be a copy of $a.ae$
  for cat in $b.attack\_evidence$ **do**
    if cat not in $a.attack\_evidence$ **then**
      $merged\_ae$.append(cat)
  return $merged\_ae$

Algorithm 3: Attack Evidence Merge — AND

Input: (1) Attack evidence of two nodes: a, b, (2) $Vuls$: The set of all the CVEs on the attack graph
Output: Attack evidence of the child node c, an AND node

Algorithm `merge_ae_AND`$(a, b, Vuls)$
  $merged\_ae$ = []
  for $cat1$ in $a.attack\_evidence$ **do**
    for $cat2$ in $b.attack\_evidence$ **do**
      /* Initialize *merged*, suppose $|Vuls| = p$ */
      $merged = \{v_1 : 0, \ldots, v_p : 0\}$
      for vul in $Vuls$ **do** $merged[vul]$ = max($cat1[vul]$, $cat2[vul]$)
      if $merged$ not in $merged\_ae$ **then**
        $merged\_ae$.append($merged$)
  return $merged\_ae$

Figure 3 illustrates an example attack graph and the corresponding attack evidences for node 1 and node 4. Since there are two attack traces to node 4 involving different vulnerabilities, the attack evidence for node 4 contains two elements, so is node 1. We compute the vulnerability evidence for each node in a forward fashion from leaf nodes to the goal nodes. Our merging algorithms are explained in Algorithm 2 and Algorithm 3 for OR and AND nodes, respectively. After getting the vulnerability evidence for each node, we can determine whether a derivation node should be included in some vulnerability's blast radius using Algorithm 4. The complete blast radius algorithm is given in Algorithm 5.

Attack evidence provides a summary of vulnerabilities involved along each attack trace to a certain node and is useful for other important problems. For example, we can use it to compute the *minimal set of vulnerabilities to patch to thwart an attack goal* defined in [40]. We can count the occurrence of each vulnerability in the attack evidence and iteratively choose the vulnerabilities in descent order of occurrence; if the current vulnerability is in the same condensed attack trace of some chosen vulnerability, then we consider the next one. The iteration stops when all of the condensed attack traces contain at least one vulnerability chosen. For another application, if the administrators have assigned numerical values as the *complexity* of each exploit, they can calculate the complexity

Algorithm 4: Determine Blast Radius

Input: (1) att_ev: attack evidences for all of the attack graph nodes, (2) $Vuls$: the map from $node\_id$ to the node's vulnerability evidence for all of the nodes

Output: The BR of each vulnerability in the attack graph

```
Algorithm determine_br(att_ev, Vuls)
    /* initialize br                                   */
    Let br be an empty map
    for vul in Vuls do
        br[vul] = ∅
    for n in att_ev do
        if n.type is OR then
            for cat in att_ev[n] do
                if sum(cat.values()) == 1 then
                    Find the key s.t. cat[key] == 1
                    br[key] = br[key] ∪ {n}
    return br
```

of each attack trace by summing the exploit complexity for each condensed attack trace.

V. Implementation

The IoTA modules are implemented in Python using 2475 LoC. Physical dependencies and exploit rules are implemented in Prolog using 1179 LoC. The framework utilizes MySQL Connector Python library [1] for database operations and MulVAL [36] for attack graph generation.

Translator. The Translator module converts IoT system configuration and vulnerabilities to Prolog clauses. The initial **system configuration** (specified in JSON format) is sent to the Translator module to generate Prolog facts. The example system configuration and the translated results are shown in Listing 9 and 10, respectively.

```
{
    "devices": [
        {"name": "D-Link Router",
         "type": "router",
         "network": ["wifi1"]
        },
        {"name": "Smartthings Hub",
         "type": "gateway",
         "network": ["wifi1", "zigbee1"]
        }
    ],
    "networks": [
        {"name": "wifi1",
         "type": "Wifi"
        },
        {"name": "zigbee1",
         "type": "Zigbee"
        }
    ]
}
```

Listing 9: Example IoT system configuration JSON file.

The above JSON file lists the device and network settings of an IoT system. The device and network names are specified by the user, while device and network types use standard names predefined.

[1] https://github.com/mysql/mysql-connector-python

Algorithm 5: Blast Radius Algorithm

Input: Attack graph G

Output: Blast radius of each vulnerability in G

```
Algorithm blast_radius(G)
    /* Generate the list of unique
       vulnerabilities                                 */
    Vuls = []
    for node in G do
        if node is a primitive fact node and node
         describes a vulnerability v then
            Vuls.append(v)
    /* Initialize attack evidence for primitive
       fact nodes, suppose |Vuls| = p                  */
    queue = []
    for node in G do
        node.cat = [{v_1 : 0, ..., v_p : 0}]
        if node is a primitive fact node and node
         describes a vulnerability v then
            find the index i such that Vuls[i] = v
            node.cat = [{v_1 : 0, ..., v_i : 1, ..., v_p : 0}]
            for child in node.children do
                if child not in queue then
                    queue.enqueue(child)
    /* Iteratively build attack evidence for all
       the nodes                                       */
    while queue.length != 0 do
        node = queue.dequeue()
        cur_ae = node.parents[0].cat
        for i in 1 to length(node.parents) do
            next_ae = node.parents[i].cat
            if node.type is AND then
                cur_ae = merge_ae_AND(cur_ae,
                 next_ae)
            else
                cur_ae = merge_ae_OR(cur_ae,
                 next_ae)
        node.cat = cur_ae
    Let att_ev be the attack evidences for all nodes
    return determine_br(att_ev, Vuls)
```

```
router(dLinkRouter).
inNetwork(dLinkRouter, wifi1).

gateway(smartthingsHub).
inNetwork(smartthingsHub, wifi1).
inNetwork(smartthingsHub, zigbee1).

wifi(wifi1).
zigbee(zigbee1).
```

Listing 10: Translated Prolog facts on system configuration.

IoT apps are first sent to the App Semantic Extractor and then translated to Prolog rules. Listing 11 is an example configuration of the SmartApp *Hall Light* explained in Section IV-C. The Translator combines parsed app semantic tuple

(Listing 7) and app configuration (Listing 11) to generate Prolog rules shown in Listing 12.

```
{
  "apps": [
    {"App name": "Light on when I come home",
     "description": "Turn on the hall light if
         there is motion and the door opens.",
     "device map": {
        "bulb": "Hue Wifi Bulb",
        "contact sensor": "Ring Contact Sensor",
        "motion sensor": "Mijia Motion Sensor"
      }
    }
  ]
}
```

Listing 11: Example IoT app configuration JSON file.

```
on(Bulb) :-
    bulb(Bulb),
    reportsMotion(MotionSensor),
    motionSensor(MotionSensor),
    open(DoorContactSensor),
    doorContactSensor(DoorContactSensor).
```

Listing 12: Prolog rule for IoT app *Hall Light: Welcome Home*.

Attack Graph Generator. The Attack Graph Generator concatenates exploit rules, indirect physical dependency rules, and the translated IoT app rules into a Prolog rule file. It also combines all the translated Prolog facts (including facts about device and network configuration and direct physical dependencies) and vulnerabilities (i.e., vulnerability existence facts and exploit model facts) into a Prolog fact file. The attack goals are also inserted into the Prolog fact file. After that, the rule and fact file are then sent to MulVAL [36] library to generate the Prolog reasoning log file and the attack graph.

VI. Evaluation

A. Dataset

To generate attack graphs and conduct analysis, we need to obtain IoT system configurations, including device instances and IoT apps installed. Though thousands of IoT apps are available, how users choose apps and device instances to install is still unknown. To the best of our knowledge, currently, there is no public dataset of IoT systems configured by different users. Such information gap has long been a challenge to IoT system security research [6], [32]. To evaluate IOTA, we generate synthetic IoT systems based on real-world IoT apps and device instances. We use a top-down approach to generate IoT systems by choosing the IoT app bundles first, as they determine the whole system's functionality. Once we have determined the IoT app bundle for the system, we create system instances by selecting device instances. To emulate the scenario where a user installs IoT devices but does not connect them to any IoT apps, we add individual IoT devices in one-third of the system instances created.

We consider SmartApps in the SmartThings Repository [2], and IFTTT applets [3] for SmartThings platform and create a pool of 532 IoT apps. We build a list of 59 smart home devices of 26 types, covering all of the device types listed on SmartThings Products List [4], from motion sensors, outlets to home appliances like TV, smart oven, etc. The devices are from 16 different platforms, all of which, except Roku, HP, and Aqara, are listed on Smartthing Partners[5]. In total, we create 37 IoT system instances. The first 18 instances are created based on the 6 app bundles used in [6] (which contains malicious apps), while the next 12 instances are generated based on 4 app bundles chosen from our app pool (which are treated as benign apps). The last 7 systems are of bigger size, with at most 50 devices, to further evaluate the scalability of our framework.

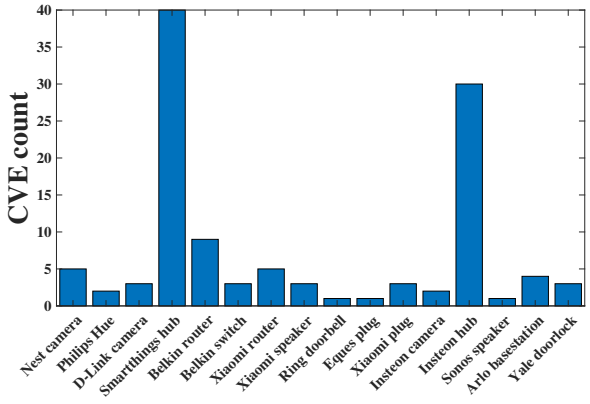


Fig. 4: The number of CVEs scanned on IoT devices.

B. Results

Vulnerability Scanning. The vulnerability scanner queries CVE database with the full name of a given IoT device instance. The scanning result is shown in Figure 4, where the devices with the largest number of CVEs are illustrated. On average, there are 7.2 CVEs per device. Figure 4 shows that device types with the largest number of detected vulnerabilities are routers, cameras, and gateways. The reason could be that due to their pivotal position in IoT systems, security researchers tend to analyze these types of devices. We manually checked all CVE records and found out that 94.2% of them are relevant to the queried device. The typical devices and their CVEs identified are listed in Table IV.

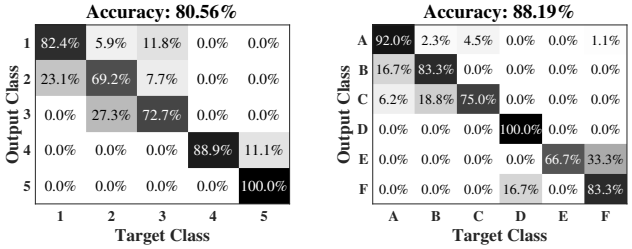
We further verified the scanned CVEs with 12 real-world IoT devices and found out 5 of them still contain vulnerabilities: we obtained the exploit scripts for Philips Wifi Bulb, D-Link DCS-5009L Camera, and Eques Elf Smart Plug and successfully launched attacks against these devices. For Wemo Insight Smart Plug and Radio Thermostat, we confirmed the existence of the vulnerabilities by matching the firmware version of devices with the one in the vulnerability reports.

Vulnerability Analysis. We ran our vulnerability analyzer on 127 CVE records of smart home IoT devices collected by the Vulnerability Scanner module and manually checked the accuracy of the predicted exploit precondition and effects.

[2]https://github.com/SmartThingsCommunity/SmartThingsPublic
[3]https://ifttt.com/search/query/smart%20home?tab=applets
[4]https://www.smartthings.com/products-list
[5]https://www.smartthings.com/partners

TABLE IV: CVEs on typical IoT devices.

Device Instance	Typical CVE(s) Scanned
Hue Wifi Bulb	CVE-2019-18980
Hue Bridge	CVE-2020-6007
Nest Cam IQ Indoor	CVE-2019-5035, CVE-2019-5036, CVE-2019-5037
D-Link DCS Camera	CVE-2019-10999
Ring Doorbell	CVE-2019-9483
Yale Lock	CVE-2019-17627
August Bridge	CVE-2019-17098
Smartthings Hub	CVE-2018-3904, CVE-2018-3917, CVE-2018-3919, CVE-2018-3925
Xiaomi Gateway	CVE-2019-15913, CVE-2019-15914
Hue Bridge	CVE-2020-6007
Arlo Basestation	CVE-2019-3949, CVE-2019-3950
Sonos Speaker	CVE-2018-11316
Xiaomi Motion Sensor	CVE-2019-15913



(a) (b)

Fig. 5: (a) Confusion matrix for exploit precondition identification. Label **1** to **5** denote preconditions listed in Table II. (b) Confusion matrix for exploit effects. Label **A** to **F** represent exploit effects listed in Table III.

The results are shown in Figure 5. Overall, our Vulnerability Analyzer achieves 80% and 88% prediction accuracy for precondition and effect, respectively. From Figure 5(a), we can see that the class `local` and `physical` have the highest accuracy, because the CVSS attack vectors for `physical` type is almost 100% accurate. And for low-power protocols, most of the time, the exploit range is local; hence, we can decide the `local` type with the help of protocol type. The precondition types with the lowest prediction accuracy are `Adjacent physically` and `Adjacent logically`. This is because some CVEs' descriptions provide vague information for these two types.

According to Figure 5(b), the most accurate class is `root`. This is because there are multiple effective indicators, such as keywords like "root", "arbitrary", etc., the CVSS subscores (confidentiality, integrity, and availability subscore all being high), and the exploit mechanism like buffer overflow, or integer overflow, etc. With these indicators combined, our prediction is accurate. The high accuracy for both the precondition and effect prediction shows our module is highly effective.

Attack Graph Generation and Analysis. Table V illustrates analysis results for 10 IoT system instances from the 37 instances we built. The first column is the ID of the system. The first four rows are the analysis results for systems built based on app bundles used in [6], and the rest of the rows are results for systems built from our own app bundles. The **# CVEs** column is the number of vulnerabilities found on the given system. We enumerate all of the system resource compromises as potential attack goals, and the **# Goals** column denotes the number of attack goals shown on the attack graph, which can be achieved by the attacker for a given system.

Table VI shows the distribution of the shortest depths of the attack traces to different goal nodes for 10 attack graphs in Table V. From the figure, the largest portion (43.9%) of the attack traces have the shortest depths among $5 \sim 8$. To evaluate the effectiveness of the attack graphs, we manually check 27 shortest attack traces whose depths are at least 9. As a result, 62.8% of the attack traces revealed by IOTA are not anticipated by the system designers.

Case study. *System 37* in Table V consists of 50 different devices, including all of the device types in Figure 4, and Wifi printer, smart TV, humidifier and toaster, etc. The vulnerability `CVE-2018-11314` identified on the Roku TV has the largest blast radius, whose cardinality is 32. By exploiting `CVE-2018-11314`, the attacker on the internet can control the smart TV and play arbitrary video. System 37 has multiple voice-related IoT apps installed, such as turning on/off the light, turning on/off the humidifier, opening the window, and locking/unlocking the door if the smart speaker receives the corresponding voice command. As a result, after compromising the TV, the attacker can control those end devices by playing videos containing the voice commands. The attacker can further compromise physical environment features such as illuminance and humidity. The blast radius of `CVE-2018-11314` directly tells system administrators about all these compromises caused by this vulnerability.

As another example, we describe the shortest attack trace to the attack goal node "opening the window" in System 28, whose depth is 18. In this example, a physically adjacent attacker first exploits `CVE-2019-17098` on the smart lock gateway to sniff the home Wifi credentials. Then he exploits `CVE-2019-3949` on the camera basestation to control the indoor camera. After that, he utilizes the rooted camera to inject the voice command "preheat the oven" into the smart home, which is sensed by the smart speaker. The speaker triggers the IoT app to start the oven. The oven may trigger smoke, which is sensed by a smoke detector. Finally, another IoT app opens the window when smoke is detected.

C. Scalability

The time and memory complexity of our framework are shown in Figure 6. From the figure we can see that, in reality, it only takes around 1.2 seconds and 120MB of memory to generate the attack graph and perform attack graph analysis for an IoT system with 50 devices. The CPU time and memory consumption grow almost linearly with the number of devices. Our graph analysis algorithms will not asymptotically increase time complexity on top of the attack graph generation algorithm because the shortest attack trace algorithm only traverses the graph once. Though the time complexity of the blast radius algorithm is bounded by the sum of the number

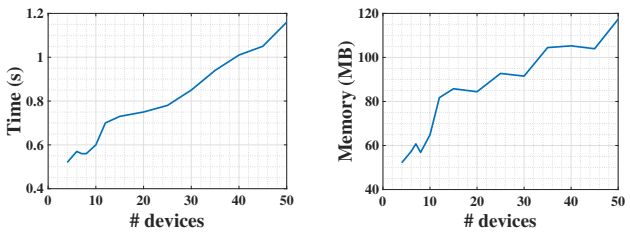
TABLE V: Attack graph analysis results on IoT system instances.

System ID	# Devices	# CVEs	# Nodes	# Edges	# Goals	Shortest depth*	CVE (|BR|)†
1	4	3	12	15	1	6	CVE-2019-18980 (4)
4	6	4	37	39	5	(2, 8)	CVE-2020-6007 (5)
8	7	4	35	44	4	(4, 10)	CVE-2019-18980 (4)
11	7	3	25	26	9	(6, 16)	CVE-2018-3904 (9)
19	10	6	36	35	4	(2, 8)	CVE-2020-6007 (4)
26	12	7	117	173	19	(4, 10)	CVE-2020-6007 (5)
28	15	9	130	182	20	(4, 18)	CVE-2019-3949 (29)
32	23	11	131	186	23	(2, 8)	CVE-2018-3904 (12)
33	31	16	209	310	23	(2, 10)	CVE-2018-3904 (19)
37	50	28	338	577	43	(2, 14)	CVE-2018-11314 (32)

*: When there are multiple attack goals in attack graph, (x, y) means the min and max depth of the shortest attack traces to different goals.
†: |BR| is the cardinality of blast radius of the CVE. This column shows the CVE with the largest blast radius cardinality.

TABLE VI: Distribution of the shortest depths for different attack goals. $\boldsymbol{d}$ means the depth of the shortest attack trace to an attack goal node.

Shortest Trace Depth	Trace Count	Percentage
$d \leq 4$	51	36.7%
$5 \leq d \leq 8$	61	43.9%
$9 \leq d \leq 12$	24	17.3%
$13 \leq d \leq 16$	2	1.4%
$d \geq 17$	1	0.7%



(a) (b)

Fig. 6: (a): CPU time vs IoT system size. (b): Memory usage vs IoT system size.

of traces to each node, in practice, this number is at the scale of $O(n^2)$ where n is the number of devices.

VII. Limitations and Future Work

Our framework uses hard-coded Prolog rules to represent direct and indirect physical dependencies between different devices. Since these rules are universal in all IoT systems, they only need to be written once and can then be copied to the Prolog rule files of all the IoT systems. And our identified six physical channels (i.e., temperature, humidity, illuminance, voice, smoke, and water) are common in IoT systems [20]. Even if new physical quantities, such as magnetic field magnitude, might be used by some special IoT systems, we can create new Prolog rules for it and insert them into all IoT systems involving such physical quantity. In the future, we plan to use machine learning to automatically extract physical channels affected/sensed by different IoT devices. We also plan to develop a vulnerability scanner for low-power IoT devices and integrate it into our IOTA framework.

VIII. Related Work

IoT security. Existing research works on IoT security focus on different parts of IoT systems. Ding et al. [20] proposed an approach to discover potential physical interactions across applications and generate interaction chains in an IoT system. Costin et al. [16] conducted a large-scale static analysis of IoT device firmware and discovered 38 previously unknown vulnerabilities. Sugawara et al. [43] explored device sensor vulnerability and presented a new class of audio injection attacks on IoT devices' microphones by converting the audio signal to laser beams. [39], [45] explored wireless communication protocol vulnerabilities. Gu et al. [25] presented an approach to sniff users' privacy by analyzing the wireless traffic. [12], [32] focused on uncovering application-level vulnerabilities using model checking techniques. Though some of the works claim to perform system-level analysis, they still just consider a subset of the core IoT components identified by our work, thus having limited capability in detecting system-level vulnerabilities.

Attack graph. Automatic attack graph construction techniques are critical for system security analysis of networked systems. There has been extensive study on attack graphs for conventional computer networks. Sheyner et al. [40] proposed automated generation of attack graphs based on symbolic model checking. But their framework suffers from the state space explosion issue, making it difficult to model systems with hundreds of hosts. [7], [35] utilized the monotonicity assumption to design attack graphs that can be generated in polynomial time. Besides, [4], [21] present methods to harden computer networks using attack graphs. Attack graphs are also applied to intrusion detection systems [11], [34].

Attack graph analyses. Ou et al. [36] introduce hypothetical analysis which answers the question of "what if there are some vulnerabilities in the system?". Sheyner et al. [40] propose two analyses, i.e., the minimal set of exploits to prevent so that the attackers fail to achieve their goals and the likelihood that the attacker will succeed. Ingols et al. [28] present automatic recommendations to improve system security by identifying a bottleneck device and patching vulnerabilities to prevent attackers from accessing the bottleneck.

Nguyen et al. [32] propose a method to attribute safety violations to either bad apps or misconfigurations.

IX. Conclusion

In this work, we design and prototype a novel framework IOTA for automatic, system-level IoT system security analysis. IOTA takes system configuration and CVE database as input and generates attack graphs showing all of the potential attack traces. Our framework further analyzes the attack graph by computing metrics, viz. the shortest attack trace and blast radius, to help system administrators evaluate vulnerabilities' impacts. Evaluation results show that IOTA is both effective (62.8% of the attack traces revealed are beyond system designers' anticipation) and highly efficient (it takes less than 1.2 seconds to analyze IoT systems of 50 devices).

Acknowledgement

This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] https://nvd.nist.gov/vuln/detail/CVE-2019-17627.
[2] https://nvd.nist.gov/vuln/detail/CVE-2019-5035.
[3] A Guide to the Internet of Things Infographic, https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html.
[4] M. Albanese, S. Jajodia, and S. Noel, "Time-efficient and cost-effective network hardening using attack graphs," in *IEEE/IFIP DSN*, 2012.
[5] Alexa Skill, https://www.amazon.com/Amazon-Key/dp/B075LY9H6H.
[6] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *Proceedings of the 29th ACM SIGSOFT ISSTA*, 2020, pp. 272–285.
[7] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *ACM CCS*, 2002.
[8] Apple HomeKit, https://developer.apple.com/homekit.
[9] August Smart Home, https://www.amazon.com/August-Home-Smart/dp/B06WW2XQ68.
[10] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.
[11] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing attack graphs for intrusion detection," in *NSPW*, 2019, pp. 16–30.
[12] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *USENIX ATC 18*, 2018.
[13] Common Vulnerabilities and Exposures (CVE), https://cve.mitre.org/.
[14] Common Vulnerability Scoring System (CVSS), https://www.first.org/cvss/.
[15] Common Weakness Enumeration (CWE), https://cwe.mitre.org/.
[16] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security*, 2014.
[17] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *ACM Asia CCS*, 2016.
[18] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," 2013.
[19] CVSS Scoring Rubrics, https://www.first.org/cvss/user-guide#Scoring-Rubrics.
[20] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *ACM CCS*, 2018.
[21] K. Durkota, V. Lisỳ, B. Bošanskỳ, and C. Kiekintveld, "Optimal network security hardening using attack graph games," in *IJCAI*, 2015.
[22] Z. Fang, H. Fu, T. Gu, Z. Qian, T. Jaeger, and P. Mohapatra, "Foresee: A cross-layer vulnerability detection framework for the internet of things," in *IEEE MASS*, 2019, pp. 236–244.
[23] Google Nest, https://developers.google.com/assistant/smarthome/.
[24] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu, and P. Mohapatra, "Iotgaze: Iot security enforcement via wireless context analysis," in *IEEE INFOCOM*, 2020, pp. 884–893.
[25] T. Gu, Z. Fang, A. Abhishek, and P. Mohapatra, "Iotspy: Uncovering human privacy leakage in iot networks via mining wireless context," in *IEEE PIMRC*, 2020.
[26] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *ACM CCS*, 2017.
[27] IFTTT SmartThings Applets, https://ifttt.com/smartthings.
[28] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *ACSAC*, 2006, pp. 121–130.
[29] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *ACL*, 2014.
[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013.
[31] National Vulnerability Database (NVD), https://nvd.nist.gov/.
[32] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "Iotsan: Fortifying the safety of iot systems," in *ACM CoNext*, 2018.
[33] No Authentication Vulnerability in Radio Thermostat, https://www.trustwave.com/en-us/resources/security-resources/security-advisories/?fid=18874.
[34] S. Noel and S. Jajodia, "Optimal ids sensor placement and alert prioritization using attack graphs," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 259–275, 2008.
[35] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *ACM CCS*, 2006.
[36] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer." in *USENIX Security*, 2005.
[37] E. Pendergrass, "Cheap, hackable iot light bulbs (or, philips bulbs have no security)," https://blog.dammitly.net/2019/10/cheap-hackable-wifi-light-bulbs-or-iot.html.
[38] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *IEEE S&P 2000*. IEEE, 2000, pp. 156–165.
[39] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *IEEE S&P*, 2017.
[40] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *IEEE S&P*, 2002.
[41] SmartThings, https://smartthings.developer.samsung.com.
[42] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security*, 2015.
[43] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu, "Light commands: laser-based audio injection attacks on voice-controllable systems," in *USENIX Security*, 2020.
[44] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, "Understanding and automatically detecting conflicting interactions between smart home iot applications," in *ESEC/FSE*, 2020.
[45] E. Y. Vasserman and N. Hopper, "Vampire attacks: draining life from wireless ad hoc sensor networks," *IEEE TMC*, vol. 12, no. 2, pp. 318–332, 2011.
[46] Vulnerability Scanning Tools, https://owasp.org/www-community/Vulnerability_Scanning_Tools.
[47] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *ACM CCS*, 2019.
[48] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in *ACM CCS*, 2018.
[49] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *USENIX Security*, 2019.