Towards System-Level Security Analysis of IoT Using Attack Graphs

Zheng Fang, Hao Fu, Tianbo Gu, Pengfei Hu, *Member, IEEE,* Jinyue Song, Trent Jaeger, *Member, IEEE,* and Prasant Mohapatra, *Fellow, IEEE*

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 or communication protocols, 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. IOTA automatically translates vulnerabilities, exploits, and device dependencies to Prolog clauses and invokes MulVAL to construct attack graphs. To evaluate vulnerabilities' system-wide impact, we propose three metrics based on the attack graph, which provide guidance on hardening 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

1 INTRODUCTION

THE last decade witnessed the rapid development and wide deployment of IoT systems. According to [1], 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 [2], Apple Homekit [3], and Google Nest [4], 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 [5], [6], [7], [8], [9].

Existing research on IoT security only focuses on a single or a subset of the IoT components. For instance, [10], [11], [12] analyze IoT device firmware, [13], [14] investigates IoT wireless protocols, and [5], [6], [15] 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 [16], but they may also compromise an indoor camera [17] and use it to inject voice, triggering a smart speaker to launch the door-open command [18]. 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 [19], [20], [21] 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* [20], [22] and *exploit-dependency attack graph* [19], [21]. 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 [19].

However, existing exploit-dependency attack graph frameworks cannot be readily applied to IoT systems due to multiple design limitations. First, existing exploit-

- *Zheng Fang and Hao Fu are with Meta Platforms, Inc., Menlo Park, CA, 94025, USA.*
  E-mail: {zhengfang,haofu}@meta.com.
- *Tianbo Gu, Jinyue Song, and Prasant Mohapatra are with the Department of Computer Science, University of California, Davis, Davis, CA 95616, USA.*
  E-mail: {zkfang,haofu,tbgu,jysong,pmohapatra}@ucdavis.edu.
- *Pengfei Hu (Corresponding author) is with School of Computer Science and Technology, Shandong University, Qingdao 266237, China.*
  E-mail: phu@sdu.edu.cn.
- *Trent Jaeger is with Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA.*
  E-mail: tjaeger@cse.psu.edu.

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 [23] 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 [24]. 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 [25], 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 [19] to generate attack graphs.

With regards to (b), we propose three novel metrics: *shortest attack trace* to an attack goal, *blast radius* of a vulnerability, and the *severity score* 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. Severity score helps us identify the most critical vulnerability to patch when the importance of each resource we want to protect is known. 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* [20]. 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 proposed metrics 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 design IOTA, a novel framework to conduct automatic, system-level IoT system security analysis and generate attack graphs showing all potential attack traces. We provide the source code of IOTA for download at https://github.com/pmlab-ucd/IOTA.
- We design formal models for IoT exploits and implement automatic translation from system configuration and vulnerability information to Prolog clauses.
- We propose three metrics to quantitatively evaluate the attack complexity (shortest attack trace) and vulnerability's system-level impacts (blast radius and severity).
- 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.

2 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 [26]. 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.

2.1 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 [10], [27], [28]. However, some vulnerabilities are found in the device's physical components [29], [30]. 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

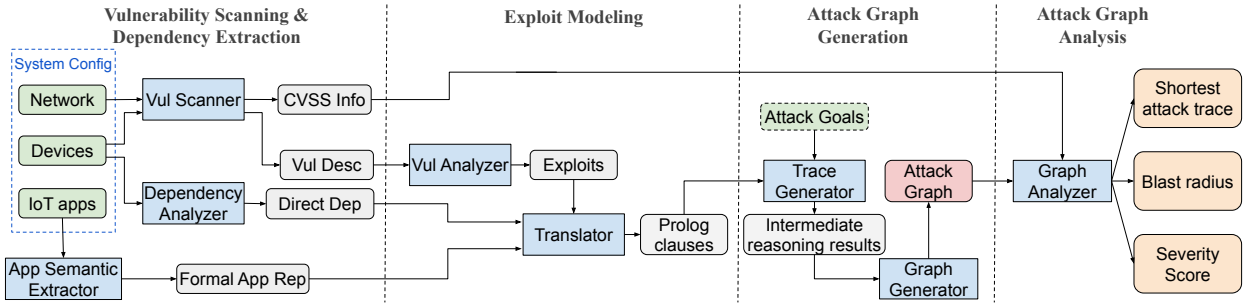


Fig. 1: IOTA pipeline. The blue boxes are IOTA modules. The green, red, and gray boxes represent input, output, and intermediate results, respectively. The *Attack Goals* can be set by the system administrator and is optional.

dependencies to compromise other devices. For example, if the attackers can control the heater, they can increase the room temperature, which will trigger a temperature sensor event.

2.2 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 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.

2.3 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 [26], [31].

2.4 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.

TABLE 1: Comparison of IOTA with other IoT security analysis frameworks.

Framework	Inter-app Analysis	Indirect Phy. Dep.	Direct Phy. Dep.	IoT CVEs	Attack Graph	Mitigation Guidance
IOTA	✓	✓	✓	✓	✓	✓
IoTSafe [32]	✓	✓				
IoTSeer [33]	✓	✓				
IoTMon [7]	✓	✓				
IoTSan [5]	✓					
Soteria [6]	✓					

3 System Overview

IOTA is a framework for automatic IoT attack graph generation and analysis. The generated attack graphs show all of the attack traces to IoT system resources that attackers can compromise. Based on the attack graphs, we further propose two metrics to provide recommendations for system hardening. Prior research on IoT system security [5], [6], [7], [32], [33] only focuses a subset of the essential IoT components, which neither uncover all of the attack traces nor provide guidance on how to prioritize patching vulnerabilities when there are dozens or hundreds of CVEs in an IoT system. In Table 1, we compare IOTA with other frameworks on identifying IoT vulnerabilities at the system level. The related work can be found in Section 7.

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) [34] 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 [19] to generate attack graphs. **Attack Graph Analysis** stage takes the generated attack graph as input and computes three metrics: the shortest attack traces to attack goal nodes and the blast radius and severity score of each vulnerability.

The input to our framework is a system configuration JSON file containing device name, device type, the network

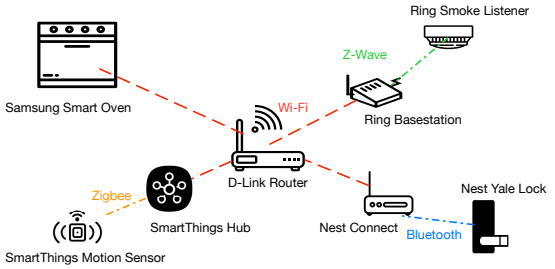


Fig. 2: IoT system example.

the device is on, and the IoT apps. A configuration example for IoT system in Figure 2 is provided in Listing 11 in Appendix A. It is IoT System 11 in Table 6, which consists of 8 devices and 4 wireless networks.

4 IOTA DESIGN

In this section, we explain the design of the IOTA modules as shown in Figure 1.

4.1 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 [35], [36] and the slow firmware upgrade rate [12].

Given the IoT devices installed and the communication protocols used, the Vulnerability Scanner module searches the CVE database [37] for vulnerabilities. We fetch the CVE JSON feeds since 2010 from the National Vulnerability Database (NVD) [38] 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 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-2018-3904` exists on `smartthingsHub`.

```
vulExists(smartthingsHub, 'CVE-2018-3904').
```

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

4.2 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 2. 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.

TABLE 2: 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

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.

For indirect physical dependencies, we identify the physical channels a device can sense and modify using NLP techniques. Google Assistant [39] lists all the *traits* a device can have for each device type. A trait consists of attributes (i.e., sensor readings) and commands (i.e., capabilities). We use Stanford CoreNLP framework [40] and Natural Language Toolkit (NLTK) [41] to parse the natural language description for each attribute and command description to extract nouns. After that, we use the Word2Vec model [42] to get the embedding of each noun and the six physical channels, and then compute cosine similarity between each noun and each physical channel. If the similarity is larger than threshold, we add corresponding physical channel to set of channels the device senses or modifies. NLP results are sent to Translator module to generate Prolog rules.

Direct physical dependencies are hard-coded as Prolog rules because they are universal in all IoT systems, regardless of the installation of certain IoT apps. Besides, since there are limited kinds of direct dependencies, hard-coding them guarantees accuracy. 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. The first Prolog rule in Listing 2 means if the oven is on, then there will be smoke, and the second rule means if there is smoke, the smoke detector will report smoke. Listing 3 means if the outlet is turned off, then any device plugged into it will also be off.

```
exists(smoke) :-
    on(Oven),
    oven(Oven).

reportsSmoke(SmokeDetector) :-
    exists(smoke),
    smokeDetector(SmokeDetector)
```

Listing 2: Indirect physical dependency.

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

```
    off(Outlet).
```

Listing 3: Direct physical dependency.

4.3 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. In the description sentence, the conditional clause represents the trigger, i.e., "if-this" part, while the main clause represents the action, i.e., "then-that" part.

We follow the techniques used in SmartAuth [43] and IoTGaze [26] and use Stanford CoreNLP framework and Natural Language Toolkit (NLTK) 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 IoT app "Unlock the door when there is smoke." is shown in Listing 4. Since both the conditional clause and the main clause contains just one simple sentence, the relationship is set to `'NONE'`.

```
conditional:  ('NONE', ['there is smoke'])
main:  ('NONE', ['Unlock the door'])
```

Listing 4: Splitting clauses into simple sentences for the IoT app *Unlock the door when there is smoke*.

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: [(['smoke'], ['is'])]
main clause: [(['the door'], ['Unlock'])]
```

Listing 6: Noun and verb phrases extracted for the IoT app *Unlock the door when there is smoke*.

Finally, we use Word2Vec model 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.

```
('NONE', ['smoke detector'], ['on'], 'NONE', ['door lock'], ['unlock'])
```

Listing 7: Internal representation of the IoT app semantic.

4.4 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 [19] 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: *Network, Adjacent physically, Adjacent locally, Local, Physical* (details in Table 8 in Appendix C). 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 [44], 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`. Like Dependency Analyzer, we use Word2Vec to compute the cosine similarity to decide whether the keyword or its synonyms are in the CVE description or not. 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 precondition to `network`.

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: *Root, Device control, Command injection, Event access, Wifi, access, DoS* (details in Table 9 in Appendix C). 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 capability of attacking other devices on 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 [45] 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 threshold), or `root privilege` (if confidentiality, integrity, and availability are all greater than the threshold).

Probability. We compute probability based on the *exploitability* subscore of the CVSS score for each CVE, which considers key factors such as attack vector, attack complexity, privileges required by the attacker, user interactions (for example, the DNS rebinding exploit requires the user to click a bait website), and attack timing (e.g., an exploit is effective only when the device is unpaired). We convert the exploitability subscore to exploit probability by rescaling the exploitability subscore to the range of 0 to 1. With exploit probabilities, we can compute the probability of each node on the attack graph using the formula given in [46]. Our graph analyzer implements probability computation, which is used as one of the building blocks of the *severity* metric defined in Section 4.6.

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` and the effect is that the attacker gets root privilege on this device. The exploit probability is 0.7.

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

Listing 8: Prolog fact for an exploit model.

4.5 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 camera from being rooted, then the attack goal is root privilege on the camera. When there are no security objectives specified, we enumerate all potential privileges attackers may get on all devices as attack goals.

Figure 3 shows a small attack graph, where node 18 and node 23 represent the attack goals: to spoof a motion sensor event and to unlock the doorlock, respectively. The meaning

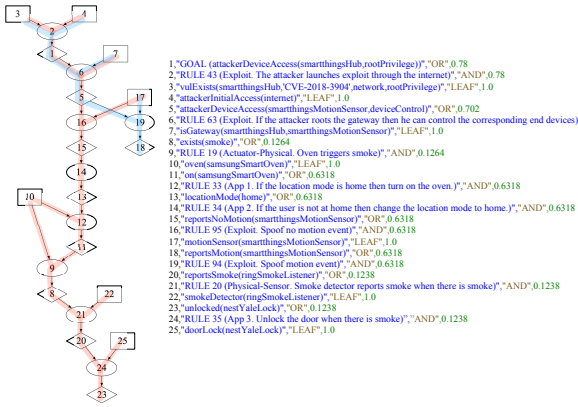


Fig. 3: IoT attack graph example. The meaning, type, and probability of each node are shown on the right.

of each node is annotated on the right of the figure. The *attack trace* (formally defined in Section 4.6) to these attack goal nodes are highlighted in red and blue. To reach node 23, the attacker first gains root privilege on the SmartThings hub from the internet. Then he spoofs a motion sensor event, and utilizes two IoT apps to turn on the oven. The oven causes smoke, which is sensed by a smoke detector. Finally, IoT App 3 unlocks the doorlock when there is a smoke alarm. For complicated attack graphs, there can be multiple attack traces to the same attack goal node.

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. A detailed explanation of attack graph structure can be found in [19].

4.6 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 3 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 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 source nodes of G' are primitive fact nodes; and (4) 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.

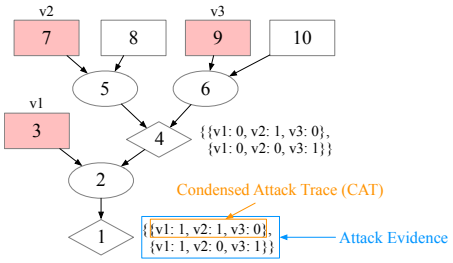


Fig. 4: Example attack graph and the corresponding attack evidence for node 1 and node 4. Node 3, 7, and 9 are primitive fact nodes describing different vulnerabilities represented as $v1$, $v2$, and $v3$.

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 (Algorithm 1 in Appendix B) 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 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.

Figure 4 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 in Appendix B. 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 in Appendix B. The complete blast radius algorithm is given in Algorithm 5 in Appendix B.

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 [20]. 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 administrators assign numerical values as the *complexity* of each exploit, they can calculate the complexity of each attack trace by summing the exploit complexity for each condensed attack trace.

Severity. Even though a vulnerability's blast radius shows all of the privileges the attacker can get by exploiting only that vulnerability, it treats these privileges equally. However, this is usually not the case in reality. For example, the attacker's being able to unlock a doorlock is much more severe than turning on a smart light bulb. Hence, we come up with the metric of severity (also called severity score) for system administrator to sort the vulnerabilities if all the attack goals in an attack graph is known. The severity score of a node is a non-negative number representing its contribution to all of the attack goal nodes. The larger a node's severity is, the more it contributes.

Our definition of severity is different from CVSS' impact score in that ours consider the impact of a vulnerability on all of the compromised system resources (represented as attack goals), whereas the latter considers the *immediate* impact. For example, if a CVE enables an attacker to get the root privilege on a device, then the CVSS' impact score represents this step, not considering any implied results, such as attacker using the rooted device to compromise another device and ultimately reaching his goal. Besides, severity scores work well in the case of physical channels because they are computed based on the generated attack graph, which contains all of the potential attacks, including both cyber attacks and physical attacks. The physical dependencies are represented as edges in the attack graph, just as cyber attacks, if they can be utilized by the attacker to compromise the final attack goal.

Computing the severity score for each node requires the following as input: (1) the probability of each node of the attack graph, and (2) the severity of attack goal nodes. The severity of a goal node can be converted from the importance of the resources it represents and is set by the administrator. For example, we set the severity of controlling a smart lock to 100, due to its significance for the smart home; in comparison, we set the severity of turning off the coffee machine to 5, since it is not important as an attack goal. The probability of each node of an attack graph is computed using [46].

The severity for each node is computed *backwards*, from attack goal nodes to the leaf nodes. Our formulas to back-propagate severity are shown in Figure 5. If node 0 is an OR node with severity S_0 (as the subfigure on the left), then the larger the probability of a parent node is, the more likely will that parent node lead to the OR node. Hence, we should assign a larger portion of S_0 to that parent node. In contrast, if node 0 is an AND node (as the subfigure on the right), all of its parent nodes must be true simultaneously. Thus the

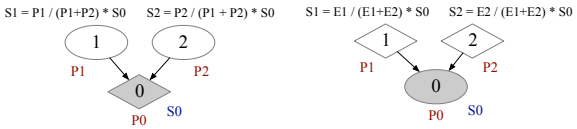


Fig. 5: The formulas to back-propagate severity scores for AND and OR nodes. P_i and E_i ($E_i = -\log P_i$) mean the probability and entropy of node i, respectively.

parent node having the smallest probability becomes the most critical condition and should get the largest portion of S_0. The formula can be generalized to the case where there are more than 2 parent nodes. Suppose node 0 has k parent nodes with probabilities P_1 to P_k, then the severity of parent node i is computed as $S_i = P_i/(P_1 + \cdots + P_k) \times S_0$. The formula for AND node can be generalized similarly. Our backward computation algorithm is summarized in Algorithm 6 in Appendix B.

5 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 [19] 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. Listing 9 shows part of the translation result of the example IoT system configuration input in Listing 11 in Appendix A.

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

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

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

Listing 9: Translated Prolog facts on system configuration.

IoT apps are first sent to the App Semantic Extractor and then translated to Prolog rules. For example, the app configuration for IoT System 11 is shown in Listing 11 in Appendix A. The Translator combines parsed app semantic tuple (Listing 7) and the app configuration to generate Prolog rules, as shown in Listing 10.

```
unlocked(DoorLock) :-
    doorLock(DoorLock),
    reportsSmoke(SmokeDetector)
```

Listing 10: Prolog rule for IoT app *Fire alarm*.

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 [19] library to generate the Prolog reasoning log file and the attack graph.

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

TABLE 3: IoT app analysis results on different IoT platforms.

IoT Platform	Total # Apps	# Apps in IFTTT Form	Percent	# Correctly Parsed	Acc*
ADT	19	10	52.6%	6	60%
Arlo	42	18	42.9%	10	55.6%
August	28	20	71.4%	13	65%
Belkin	27	16	59.3%	11	68.8%
Hue	37	19	51.4%	13	68.4%
Nest	84	67	79.8%	45	67.2%
SmartThings	184	63	34.2%	46	73%
Total	421	213	50.6%	144	67.6%

*: Acc = #Correctly Parsed/#Apps in IFTTT Form

6 Evaluation

6.1 Dataset

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 [47], and IFTTT applets [48] for SmartThings platform and create a pool of 421 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 [49], 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 [50]. In total, we create 37 IoT system instances. The first 18 instances are created based on the 6 app bundles used in [51] (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.

6.2 Results

App Semantic Extraction. To evaluate our NLP-based IoT app semantic extractor, we collected 421 IoT apps from 7 IoT platforms and ran our app semantic extractor module on them. The results are shown in Table 3. From the table, we can see that, on average, 50.6% of the IoT apps are described using if-this-then-that form. And for the apps that are in *if-this-then-that* form, our module achieves 67.6% accuracy. This is because even though some apps' descriptions are in *if-this-then-that* form, they are either too general or ambiguous. For example, there are 17 such cases in SmartThings' SmartApps and here are some of them: "Receive notifications when anything happens in your home.", "Trigger Simple Control activities when certain actions take place in your home.", and "Play or pause your Speaker when certain actions take place in your home."

Dependency Analyzer. For indirect physical dependency between different devices, our dependency analyzer module automatically detects the physical channels each device senses or modifies. We evaluate the effectiveness of

TABLE 4: Physical channels identified for different device types. **T**: temperature, **H**: humidity, **I**: illuminance, **V**: voice, **S**: smoke, **W**: water. ◯ represents channels identified by our module, while ✓ represents channels confirmed by manual examination.

Device Type	T	H	I	V	S	W
AC	◯ ✓					
Camera		✓		◯ ✓		
Heater	◯ ✓	✓				
Humidifier		◯ ✓				
Light Bulb	◯		◯ ✓			
Smoke Detector					◯ ✓	
Sprinkler						◯ ✓
Speaker				◯ ✓		
TV				◯ ✓		

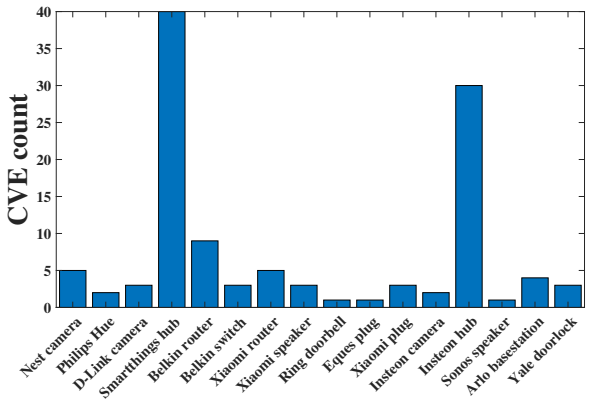


Fig. 6: The number of CVEs scanned on IoT devices.

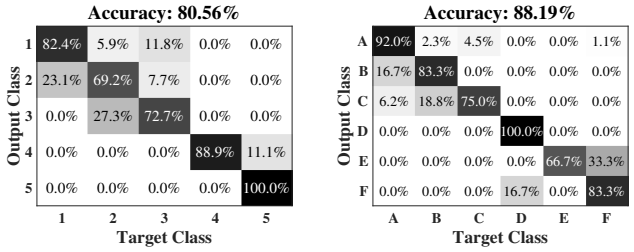
the module using Google Home device descriptions [39], which specify the traits for each device type. We crawl the natural language descriptions of the traits and capabilities associated with different device types and run our module to identify the physical channels they interact with. The results are listed in Table 4. From the table, we can see that 7 out of the 11 physical channels are correctly predicted for different devices, except AC, heater, and light bulb. The AC and heater missed the humidity because there is no such keywords related to humidity in the trait descriptions. The light bulb has temperature as one of the channels it modifies. This is because the description of the `ColorSetting` trait uses the word "temperature" to represent color temperature.

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 6, where the devices with the largest number of CVEs are illustrated. On average, there are 7.2 CVEs per device. Figure 6 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 5.

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.

TABLE 5: 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
Arlo Basestation	CVE-2019-3949, CVE-2019-3950
Sonos Speaker	CVE-2018-11316
Xiaomi Motion Sensor	CVE-2019-15913



(a) (b)

Fig. 7: (a) Confusion matrix for exploit precondition identification. Label **1** to **5** denote preconditions listed in Table 8 in Appendix C. (b) Confusion matrix for exploit effects. Label **A** to **F** represent exploit effects listed in Table 9 in Appendix C.

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. The results are shown in Figure 7. Overall, our Vulnerability Analyzer achieves 80% and 88% prediction accuracy for precondition and effect, respectively. From Figure 7(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 7(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.

We randomly picked 10 IoT vulnerabilities and compare the vulnerability analysis results and the groundtruth. We get the groundtruth by manually checking the description, CVSS score, and the references of the vulnerability on the NVD-CVE website [38]. The results are shown in Table 10 in Appendix C.

Attack Graph Generation and Analysis. Table 6 illus-

TABLE 6: Attack graph analysis results on IoT system instances.

Sys ID	# Devices	# CVEs	# Nodes	# Edges	# Goals	Shortest depth*	CVE (|BR|)†	CVE (Severity)‡	CVE (CVSS)§
1	4	3	12	15	1	6	CVE-2019-18980 (4)	CVE-2019-18980 (20)	CVE-2020-8864 (8.8)
4	6	4	37	39	5	(2, 8)	CVE-2020-6007 (5)	CVE-2020-6007 (27.2)	CVE-2019-15913 (9.8)
8	7	4	35	44	4	(4, 10)	CVE-2019-18980 (4)	CVE-2019-17098 (51.2)	CVE-2020-8864 (8.8)
11	7	3	25	26	9	(6, 16)	CVE-2018-3904 (9)	CVE-2018-3904 (115)	CVE-2018-3904 (9.9)
19	10	6	36	35	4	(2, 8)	CVE-2020-6007 (4)	CVE-2019-17627 (100)	CVE-2018-3904 (9.9)
26	12	7	117	173	19	(4, 10)	CVE-2020-6007 (5)	CVE-2019-17098 (166.9)	CVE-2019-5035 (9.0)
28	15	9	130	182	20	(4, 18)	CVE-2019-3949 (29)	CVE-2019-3949 (136.5)	CVE-2019-3949 (9.8)
32	23	11	131	186	23	(2, 8)	CVE-2018-3904 (12)	CVE-2018-11314 (136.8)	CVE-2018-3904 (9.9)
33	31	16	209	310	23	(2, 10)	CVE-2018-3878 (19)	CVE-2018-11314 (139.2)	CVE-2018-3878 (9.9)
37	50	28	338	577	43	(2, 14)	CVE-2018-11314 (32)	CVE-2019-5035 (100.6)	CVE-2018-3878 (9.9)

*: 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.
‡: CVE (Severity) is the CVE with the highest severity score. We specify initial severity scores for different attack goals ourselves.
§: CVE (CVSS) is the CVE with the highest CVSS score(s) in the system.

TABLE 7: 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%

trates 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 [51], 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. From Table 6, we can see that for many IoT systems, the vulnerabilities with the highest CVSS scores are not the same as the ones with the largest blast radius or severity score. This is because the CVSS score measures the local impact of the vulnerability, without the implied effects, while our metrics consider the impacts on all the IoT resources of a system. For example, in IoT System 19, the CVSS score of `CVE-2019-17627` is 6.5, which is much less than that of `CVE-2018-3904`. However, since it enables the attacker to unlock the doorlock, it has the largest severity score, as expected.

Table 7 shows the distribution of the shortest depths of the attack traces to different goal nodes for 10 attack graphs in Table 6. 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.

6.3 Case Study

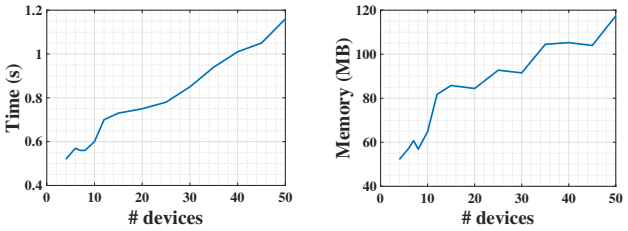
Shortest Attack Trace. The shortest attack trace to the attack goal node "opening the window" in System 28 has a depth of 18. In this example, a physically adjacent attacker first exploits `CVE-2019-17098` on the smart lock gateway to sniff the home Wifi credentials. Then he exploits `CVE-2019-3949` 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.

Blast Radius. System 37 consists of 50 different devices, including all of the device types in Figure 6, 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.

Severity. System 19 consists of 10 devices, including contact sensor, temperature sensor, doorlock, smart bulb, smart speaker, and gateways. According to Table 6, even though `CVE-2020-6007` has the largest blast radius, which contains 4 privileges which can be obtained by the attacker, these privileges are all about controlling the bulb and the room's illuminance. Because the system administrator assigns small severity to these compromises, the severity of `CVE-2020-6007` is 60.98. In comparison, `CVE-2019-17627`'s blast radius only contains attacker's controlling the doorlock. However, since this compromise is treated as severe, the administrator assigns 100 as its severity score. And the severity score gets back-propagated to `CVE-2019-17627`. We can see that severity scores can help the administrator evaluate the system-level impact of each vulnerability, thus prioritizing the fix.

6.4 Scalability

The time and memory complexity of our framework are shown in Figure 8. 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



(a) (b)

Fig. 8: (a): CPU time vs IoT system size. (b): Memory usage vs IoT system size.

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 and severity score algorithm only traverse the graph once. Though the time complexity of the blast radius algorithm is bounded by the sum of the number of traces to each node, in practice, this number is at the scale of $O(n^2)$ where n is the number of devices.

7 Related Work

IoT Security. Existing research works on IoT security focus on different parts of IoT systems. Ding et al. [7] proposed an approach to discover potential physical interactions across applications and generate interaction chains in an IoT system. Costin et al. [10] conducted a large-scale static analysis of IoT device firmware and discovered 38 previously unknown vulnerabilities. Sugawara et al. [30] 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. [13], [14] explored wireless communication protocol vulnerabilities. Gu et al. [52] presented an approach to sniff users' privacy by analyzing the wireless traffic. [5], [6] 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.

IoT Physical Vulnerabilities. Different from traditional networked systems, IoT devices' interaction via the physical channels brings additional vulnerabilities. Recently, there are several works on identifying unexpected IoT devices' physical interaction. IoTMon [7] uses static analysis and NLP to identify physical channels and detects risky inter-app interaction chains. IoTSafe [32] is a run-time safety and security policy enforcement system, focusing on physical interaction among IoT devices. IoTSeer [33] combines static analysis on IoT apps and dynamic analysis with security policies to detect policy violations due to physical channels. Since physical interaction depends on devices' placement and the environment, dynamic systems can detect IoT physical dependencies more accurately. However, they did not consider other IoT vulnerabilities, such as sensors' firmware or hardware vulnerability, which may enable the attacker to bypass the enforcement system. For example, the attacker can directly compromise the temperature sensor and send a fake temperature reading while turning on the heater to make the room temperature above the threshold. Besides, these frameworks require IoT apps' source code for static analysis, while our dependency analyzer only requires access to IoT devices' descriptions. Our framework also identified an important channel, i.e., voice, which are missed by existing frameworks.

NLP for IoT Security. NLP techniques have been used in IoT app analysis [7], [26], [31], [43]. Ding et al. [7] utilizes NLP to extract entity keywords for physical channels from IoT app descriptions and to cluster these entity keywords based on their Word2Vec similarities. Tian et al. [43] use NLP to automatically extract trigger and action information from apps' descriptions and annotations. Then they compare the app semantics with the actual app behavior obtained from static analysis to detect overprivilege problems in IoT apps. [26], [31] monitor IoT apps by utilizing NLP to extract IoT app semantics and comparing it to the actual app behavior sniffed from the wireless traffic. Though our IoT app analysis using NLP is similar to those works, to the best of our knowledge, IOTA is the first framework which applies NLP techniques to analyzing CVEs and inferring IoT devices' indirect physical dependencies.

Attack Graphs. Automatic attack graph construction techniques are critical for system security analysis of networked systems. Sheyner et al. [20] 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. [19], [21] utilized the monotonicity assumption to design attack graphs that can be generated in polynomial time. Ingols et al. [53] present automatic recommendations to improve system security by identifying a bottleneck device and patching vulnerabilities to prevent attackers from accessing the bottleneck. [54], [55] presented methods to harden computer networks using attack graphs.

8 Discussion & Limitations

Manual Effort Required. It has long been a challenge to evaluate frameworks on IoT system security [5], [51] due to the lack of real-world IoT dataset. Though there are thousands of IoT apps 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. To evaluate our framework, we designed a top-down approach to generate synthetic IoT systems based on real-world IoT devices and apps. In addition, there is no IoT vulnerability dataset we can use directly to evaluate Vulnerability Analyzer module. Even though the CVE [37] database contains all the known vulnerabilities, including IoT vulnerabilities, there are no preconditions and effects designed specifically for IoT attacks, as shown in Table 8 and Table 9 in Appendix C. For example, the CVSS score's *attack vector* does not differentiate whether the attackers should be on the Wifi network (`wifi:adjacent_logically`), or they just need to be within Wifi's radio range (`wifi:adjacent_physically`). As a result, we have to manually check the results of the Vulnerability Scanner, Vulnerability Analyzer, and IoT Semantic Extractor module. However, these manual efforts are for evaluation only; once the framework is deployed, there's no need to manually check them.

NLP for IoT App Analysis. Even though the constituency parser [40] used in our framework is widely used in various research fields, including IoT security [7], [26], [31], [43], it is very sensitive to perturbations — the parse tree is constructed differently whether there is a comma or not, even though for humans the semantics are the same. This can affect the splitting results of the IoT app description sentences. Besides, the performance of matching noun phrases to device types using Word2Vec cosine similarity can also be improved. One potential solution is to use device annotations as additional information. Since the annotations are in the user interface, the matching still requires no access to the apps' source code. Moreover, we can match noun phrases to device types using different thresholds for different devices.

9 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, blast radius, and severity score, to help system administrators evaluate vulnerabilities' impacts. We create 37 synthetic smart home IoT systems based on 532 real-world IoT apps and 59 smart home devices and utilize our framework to analyze their security. 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).

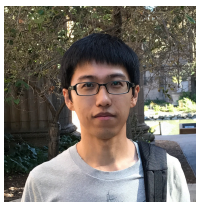
Acknowledgments

This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA), the National Natural Science Foundation of China (No. 62202276), and the Shandong Science Fund for Excellent Young Scholars (No. 2022HWYQ-038).

References

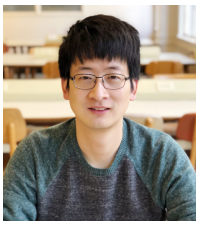
[1] A Guide to the Internet of Things Infographic, https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html.
[2] SmartThings, https://smartthings.developer.samsung.com.
[3] Apple HomeKit, https://developer.apple.com/homekit.
[4] Nest, https://developers.google.com/assistant/smarthome/.
[5] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "Iotsan: Fortifying the safety of iot systems," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 191–203.
[6] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 147–158.
[7] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 832–846.
[8] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1439–1453.
[9] 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 *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2019.
[10] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 95–110.
[11] 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019.
[12] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *20th Network & Distributed System Security Symposium (NDSS)*, 2013.
[13] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 195–212.
[14] E. Y. Vasserman and N. Hopper, "Vampire attacks: draining life from wireless ad hoc sensor networks," *IEEE transactions on mobile computing*, vol. 12, no. 2, pp. 318–332, 2011.
[15] 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 *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1215–1227.
[16] https://nvd.nist.gov/vuln/detail/CVE-2019-17627.
[17] https://nvd.nist.gov/vuln/detail/CVE-2019-5035.
[18] Alexa Skill, https://www.amazon.com/Amazon-Key/dp/B075LY9H6H.
[19] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 336–345.
[20] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 273–284.
[21] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
[22] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 2000, pp. 156–165.
[23] Vulnerability Scanning Tools, https://owasp.org/www-community/Vulnerability_Scanning_Tools.
[24] X. Ou, S. Govindavajhala, A. W. Appel *et al.*, "Mulval: A logic-based network security analyzer." in *USENIX security symposium*, vol. 8. Baltimore, MD, 2005, pp. 113–128.
[25] IFTTT SmartThings Applets, https://ifttt.com/smartthings.
[26] 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-IEEE Conference on Computer Communications*. IEEE, 2020.
[27] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 437–448.
[28] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.
[29] 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 *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 881–896.
[30] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu, "Light commands: Laser-based audio injection attacks on voice-controllable systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2631–2648.
[31] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1074–1088.
[32] W. Ding, H. Hu, and L. Cheng, "Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery," in *the 28th Network and Distributed System Security Symposium (NDSS 2021)*, 2021.
[33] M. O. Ozmen, X. Li, A. Chu, Z. B. Celik, B. Hoxha, and X. Zhang, "Discovering iot physical channel vulnerabilities," in

ACM SIGSAC Conference on Computer and Communications Security (CCS), 2022.
[34] Common Vulnerability Scoring System (CVSS), https://www.first.org/cvss/.
[35] 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.
[36] No Authentication Vulnerability in Radio Thermostat, https://www.trustwave.com/en-us/resources/security-resources/security-advisories/?fid=18874.
[37] Common Vulnerabilities and Exposures, https://cve.mitre.org/.
[38] National Vulnerability Database (NVD), https://nvd.nist.gov/.
[39] Google Assistant, https://developers.google.com/assistant/smarthome/guides.
[40] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL)*, 2014.
[41] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.
[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.
[43] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 361–378.
[44] Forum of Incident Response and Security Teams, Inc., "CVSS Scoring Rubrics," https://www.first.org/cvss/user-guide#Scoring-Rubrics.
[45] Common Weakness Enumeration (CWE), https://cwe.mitre.org/.
[46] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, "An attack graph-based probabilistic security metric," in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2008.
[47] SmartThings Public GitHub Repo, https://github.com/SmartThingsCommunity/SmartThingsPublic.
[48] IFTTT Smart Home Applets, https://ifttt.com/search/query/smart%20home?tab=applets.
[49] SmartThings Products List, https://www.smartthings.com/products-list.
[50] SmartThings Partners, https://www.smartthings.com/partners.
[51] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 272–285.
[52] T. Gu, Z. Fang, A. Abhishek, and P. Mohapatra, "Iotspy: Uncovering human privacy leakage in iot networks via mining wireless context," in *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE, 2020.
[53] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 121–130.
[54] M. Albanese, S. Jajodia, and S. Noel, "Time-efficient and cost-effective network hardening using attack graphs," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.
[55] K. Durkota, V. Lisỳ, B. Bošanskỳ, and C. Kiekintveld, "Optimal network security hardening using attack graph games," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

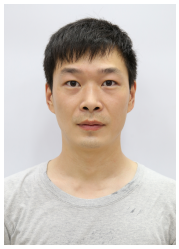


Zheng Fang received his B.Eng. and M.S. degree from Huazhong University of Science and Technology, and Columbia University, respectively. He received his Ph.D. degree in Computer Science at University of California, Davis. His research interests include internet of things (IoT), security, and program analysis. He published papers in INFOCOM, IoTDI, and MASS, etc. He served as a reviewer for ICDCS, IEEE Internet of Things Journal, and IEEE Transactions on Wireless Communications, etc.

Hao Fu received his Ph.D. in Computer Science from the University of California, Davis, USA, in 2019. He was supervised by Dr. Prasant Mohapatra. His research interests include network and systems security, machine learning, and program analysis. He published more than 10 papers in relevant reputed conferences, including INFOCOM, SECON, CoNext, MILCOM, etc. He was also a reviewer of CCS, CNS, ACNS, MILCOM, etc.



Tianbo Gu received the B.S. degree from Hangzhou Dianzi University, China, in 2010, and the M.S. degree from University of Science and Technology of China, in 2013. In 2012, he received National Scholarship which is the top award for graduate students in China. He got his Ph.D. degree in Computer Science at University of California, Davis in 2020. His research interests include Internet of Things, edge computing, and smart sensing.



Pengfei Hu is a professor in School of Computer Science and Technology at Shandong University, China. He received Ph.D. in Computer Science from UC Davis. His research interests are in the areas of IoT security, AI security, mobile computing. He has published over 30 papers in premier conferences and journals on these topics, e.g. IEEE S&P, ACM CCS, IEEE INFOCOM, IEEE TMC, etc. He also served as TPC for numerous prestigious conferences and associate editors for IEEE TWC and IoTJ.

Jinyue Song received his B.S. degree from McMaster University, Canada in 2017 and his M.S. degree from the University of San Francisco in 2019. He is pursuing a Ph.D. degree at the University of California, Davis, supervised by Professor Prasant Mohapatra. His research interests cover network security, mmWave communication in vehicular networks, edge computing, and blockchain.

Trent Jaeger received the M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1993 and 1997, respectively, and spent nine years at IBM Research prior to joining Penn State. He is a Professor with the Computer Science and Engineering Department, Pennsylvania State University, where he is the Co-Director of the Systems and Internet Infrastructure Security Laboratory. He has published over 100 refereed papers on these topics and the book Operating Systems Security, which examines the principles behind secure operating systems designs. He has made a variety of contributions to open source systems security, particularly to the Linux Security Modules framework, SELinux, integrity measurement in Linux, and the Xen security architecture. His research interests include systems security and the application of programming language techniques to improve security. He was the Chair of the ACM Special Interest Group on Security, Audit, and Control.

Prasant Mohapatra is serving as the Vice Chancellor for Research at University of California, Davis. He is also a Professor in the Department of Computer Science and served as the Dean and Vice-Provost of Graduate Studies at University of California, Davis during 2016-18. He was the editor-in-chief of the IEEE Transactions on Mobile Computing. He has served on the editorial boards of the IEEE Transactions on Computers, the IEEE Transactions on Mobile Computing, the IEEE Transaction on Parallel and Distributed Systems, the ACM Journal on Wireless Networks, and Ad Hoc Networks. He is a fellow of the IEEE and a fellow of the AAAS.