Charting the Attack Surface of Trigger-Action IoT Platforms

Qi Wang,[†*] Pubali Datta,[†*] Wei Yang,[‡] Si Liu,[†] Adam Bates,[†] Carl A. Gunter[†]
[†] University of Illinois at Urbana-Champaign, [‡]The University of Texas at Dallas
{qiwang11,pdatta2,siliu3,batesa,cgunter}@illinois.edu,wei.yang@utdallas.edu

ABSTRACT

Internet of Things (IoT) deployments are becoming increasingly automated and vastly more complex. Facilitated by programming abstractions such as trigger-action rules, end-users can now easily create new functionalities by interconnecting their devices and other online services. However, when multiple rules are simultaneously enabled, complex system behaviors arise that are difficult to understand or diagnose. While history tells us that such conditions are ripe for exploitation, at present the security states of trigger-action IoT deployments are largely unknown.

In this work, we conduct a comprehensive analysis of the interactions between trigger-action rules in order to identify their security risks. Using IFTTT as an exemplar platform, we first enumerate the space of *inter-rule vulnerabilities* that exist within trigger-action platforms. To aid users in the identification of these dangers, we go on to present *i*Ruler, a system that performs Satisfiability Modulo Theories (SMT) solving and model checking to discover inter-rule vulnerabilities within IoT deployments. *i*Ruler operates over an abstracted information flow model that represents the attack surface of an IoT deployment, but we discover in practice that such models are difficult to obtain given the closed nature of IoT platforms. To address this, we develop methods that assist in inferring trigger-action information flows based on Natural Language Processing. We develop a novel evaluative methodology for approximating plausible real-world IoT deployments based on the installation counts of 315,393 IFTTT applets, determining that 66% of the synthetic deployments in the IFTTT ecosystem exhibit the potential for inter-rule vulnerabilities. Combined, these efforts provide the insight into the real-world dangers of IoT deployment misconfigurations.

CCS CONCEPTS

• **Security and privacy** → *Formal methods and theory of security*; *Vulnerability scanners*; Software security engineering; • **Computing methodologies** → *Natural language processing*; • **Computer systems organization** → *Embedded and cyber-physical systems.*

KEYWORDS

Trigger-Action IoT Platform; Inter-rule Vulnerability; Formal Methods; NLP; Information Flow

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3345662

ACM Reference Format:
Qi Wang,[†*] Pubali Datta,[†*] Wei Yang,[‡] Si Liu,[†] Adam Bates,[†] Carl A. Gunter[†]. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *2019 ACM SIGSAC Conference on Computer& Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3345662

1 INTRODUCTION

The Internet of Things (IoT) is growing rapidly. With predictions of 20 billion deployed IoT devices by 2020 [1], the IoT has evolved from isolated single devices to integrated platforms that facilitate interoperability between different devices and online services (e.g., Gmail). Samsung's SmartThings[11], Apple's HomeKit [4], IFTTT [5] and Zapier [17] are just a few examples. IoT platforms support end-user customizations, with many going so far as to provide programming frameworks for the design of simple automation logic that enable customized functionality. Currently, trigger-action programming (TAP) is the most commonly-used model to create automations in IoT. Studies have shown that about 80% of the automation requirements of typical users can be represented by TAP and that even non-programmers can easily learn this paradigm [85].

Unfortunately, as IoT deployments grow in complexity, so do their attack surface – as users further automate their homes, unexpected interactions between the automation rules may give rise to alarming new classes of security issues [81]. Consider the possibility that a user has installed the rule *If temperature exceeds 30 °C, then open my windows*; while this may be innocuous in isolation, it could be leveraged by an attacker to gain physical entry to the house if the user has also installed the rule *(If you say) "Alexa, trigger heater", then turn the heater on.* While IoT presents a variety of novel security challenges, the threats created by the ease of trigger-action automation are worthy of careful consideration.

Reasoning about the security of trigger-action IoT platforms requires a precise understanding of the interplay between trigger-action rules. The circumstances under which the interactions between two rules should be designated as a bug or vulnerability, as opposed to a *feature*, are not presently clear. Even among small rulesets, such as the real-world example shown in Figure 1, it is not immediately obvious whether this composition of 5 rules could lead to a breach in the user's home security system; in fact, because the three rules ($r2$, $r4$, $r5$) all modify the security mode of the user's *Somfy Home Security System*, there is a legitimate risk that the system could reach an unsafe state. What further frustrates analysis is the fact that trigger-action IoT ecosystems are closed-sourced and developed by a variety of third parties, rendering existing program analysis techniques unusable.

In this work, we describe three distinct and inter-related efforts to enable precise reasoning about IoT security postures. To

[*] Joint first authors.

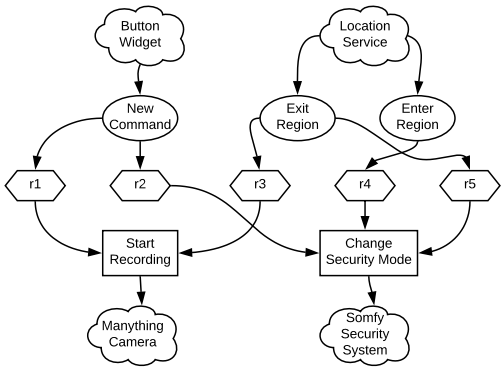


Figure 1: Interaction of rules between popular home security services from real-world examples [6]. Rules are represented as hexagon vertices, triggers using oval vertices, actions using rectangle vertices, and services using cloud vertices.

better understand trigger-action rule bugs, we first exhaustively explore the space of *inter-rule vulnerabilities* within trigger-action IoT platforms. This taxonomy of inter-rule vulnerabilities attempts to systematize problems identified by other recent work in this space [30, 32, 52, 70] and uncovers new subclasses of this vulnerability. Second, we leverage formal methods to enable the detection of these bugs; we present the design and implementation of *i*Ruler, an IoT analysis framework that leverages Satisfiability Modulo Theories (SMT) solving and model checking to discover inter-rule vulnerabilities. However, *i*Ruler requires an *information flow graph* of the IoT deployment to operate, which at present is unavailable due to the opacity of commodity IoT platforms. To overcome this obstacle in the absence of viable program analysis techniques, the third and final element of our design is an approach to infer inter-rule information flows by using Natural Language Processing (NLP) to inspect the text descriptions of triggers and actions on the IoT platform website.

We evaluate *i*Ruler against a real-world dataset of 315,393 applets found on the IFTTT website. Testing against a manually-coded ground truth of inter-rule flows, we find that our NLP tool is able to eliminate 72% of false dependencies in the IFTTT ecosystem with minimal Type I error, the sources of which we characterize in discussion. *i*Ruler detects vulnerabilities in specific *configurations* of IoT deployments, but at present robust data on realistic configurations is not publicly available. To address this, we develop a method for synthesizing plausible rulesets based on publicly-visible install counts of IFTTT applets. By testing *i*Ruler on these synthetic configurations, we discover the widespread potential for inter-rule vulnerabilities in the IFTTT platform, with 66% of the rulesets being associated with at least one such vulnerability.

2 BACKGROUND

2.1 Trigger-action IoT Platforms

Home automation IoT platforms commonly use the trigger-action programming paradigm, which provides an intuitive abstraction for non-technical users wishing to automate their devices. Broadly,

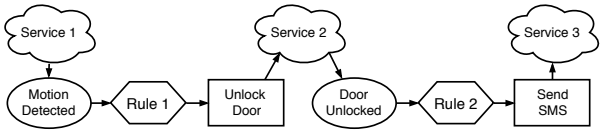
Table 1: A comparison of several popular trigger-action platforms, which vary in their support for conditions, rules with multiple actions, parameter passing from triggers to actions, and a rule store.

Platform	Support Conditions	Multiple Actions	Trigger Values used in Actions	Rule Store
SmartThings [11]	✓	✓	✓	✓
IFTTT [5]	✓	✓	✓	✓
openHAB [10]	✓	✓	✓	✓
Microsoft Flow [8]	✓	✓	✓	✓
Zapier [17]	✗	✓	✓	✓
HomeKit [4]	✗	✗	✗	✗
Iris [7]	✗	✗	✗	✓
Wink [15]	✗	✗	✗	✗

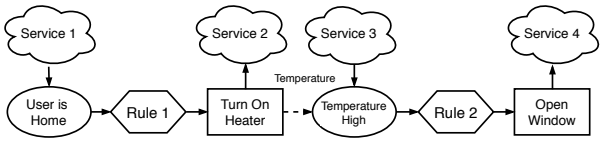
a trigger-action (TA) program specifies that when a certain trigger event occurs (e.g., motion is detected), one or more actions (e.g., turn on the light) should be subsequently executed. Emerging trigger-action models are also becoming more expressive through the introduction of advanced features. In Table 1, we compare the trigger-action models in 5 popular smart home platforms and 3 popular task automation platforms. While we note the differences between these platforms, our study considers a generalized trigger-action model in which each rule can have one trigger, one or more actions, and a condition associated with each action.

Trigger-action Rule Chaining. The power of the trigger-action programming paradigm is that rules can be chained together [81]; the execution of an action can invoke another trigger event, causing another rule to execute. There are two ways rules can be chained, examples of which are given in Figure 2 in the form of *trigger-action graphs*: rules A and B are *Explicitly Chained* if (1) A's action and B's trigger belong to the same service and (2) executing A's action directly satisfies B's trigger event; rules A and B are *Implicitly Chained* if (1) A's action and B's trigger connect to a global shared medium or state and (2) executing A's action manipulates the shared medium such that B's trigger is satisfied.

The IFTTT Platform. *If-this-then-that* (IFTTT) [5] is a web-based task-automation platform which allows users to connect different *services* to create automations using the trigger-action paradigm. Services are typically published by third parties, facilitating interoperability with smart devices (e.g., Nest thermostat) or online services (e.g., Gmail and Facebook). Each supported service publishes a set of triggers and actions that are akin to a service API. A *trigger* is a source of events in a service. For example, a trigger in the Nest thermostat service is "Temperature drops below", which fires every time the temperature drops below a threshold. An *action* is a task that a service can perform, e.g., sending an email. An *applet* (i.e., a rule) is an automation program that consists of one trigger and one or more actions. For example, a user can create an applet to send an email if the temperature drops below a threshold. Most triggers, like the one above, have *trigger fields* that determine under what circumstances the trigger event should occur. Similarly, most actions have *action fields* which are the parameters of the action. Each trigger also has *ingredients* (i.e., parameters) which are basic data available from the corresponding trigger event. For example, the subject and the sender's email address are two ingredients of an email trigger. In an applet, trigger ingredients can be used as part of a parameter by an action. An applet developer can also set further conditions on the invocation of an action by using the *filter code* feature, which adds extra flexibility in the form of a TypeScript [14]



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



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

Figure 2: Trigger-action graphs depicting (a) explicit chaining and (b) implicit chaining. Solid and dotted-line edges represent explicit and implicit chains, respectively.

code snippet. The filter code has access to the data returned by the trigger and metadata like the current time. It can use the information to override action field values or skip an action. An example filter code snippet is provided in Appendix A.

2.2 Model Checking and Rewriting Logic

Model checking [48] is a technique that checks if a system meets a given specification by systematically exploring the system's state. In an ideal case, a model checker exhaustively examines all possible system states to verify if there is any violation of specifications.

Rewriting logic [61], a logic of concurrent change that can naturally deal with state and with concurrent computations, offers a clean-yet highly expressive-mathematical foundation to assign formal meaning to open system computation. In rewriting logic, concurrent computations are axiomatized by (possibly conditional) rewrite rules of the form $l \rightarrow r$, meaning that any system state satisfying the pattern l will be transited to a system state satisfying the pattern r. For any given state, many rewrite rules can be active, thus allowing for non-determinism. Rewriting logic has been used to model and analyze different distributed systems [54–57].

3 THREAT MODEL & ASSUMPTIONS

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

- *Exploitation:* An adversary discovers an exploitable interaction between two or more benign apps or invokes a trigger event through manipulation of a 3rd party service [41].
- *Targeted Rules:* An adversary tricks a user into installing rules that enable an attack, e.g., through phishing or social engineering.
- *Malicious Apps:* An adversary develops and distributes a malicious app that contains hidden functionality [23, 38, 49, 84].

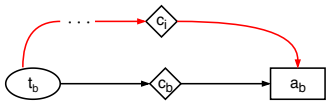


Figure 3: The condition bypass vulnerability. Two paths exist from t_b to a_b and $c_i \neq c_b$. The red line shows a rule chain to bypass c_b.

Recent work has considered powerful adversaries that obtain root access to devices [3] or compromise communication protocols [2], which are out of scope in this work. While important, these strong adversarial models run the risk of downplaying the potential dangers posed by everyday attackers without advanced technical knowledge. Prior work has demonstrated that IoT end users often make errors in writing trigger-action rules [46, 68, 86]. Since they are often unaware of the implications of rules interactions, it stands to reason that users' creation, deletion, or misconfiguration of rules leads to security vulnerabilities in their homes. Our threat model also accounts for the safety risks of benign misconfigurations, which pose a real-world threat. We thus argue that rule-level attacks are an important consideration for IoT security, and note also that similar threat models have appeared in related work [23, 30, 49, 70, 87].

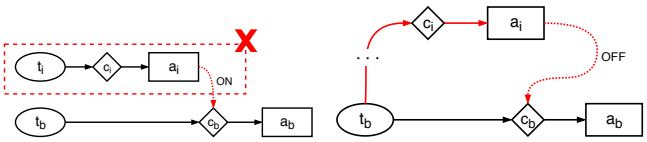
4 INTER-RULE VULNERABILITIES

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

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

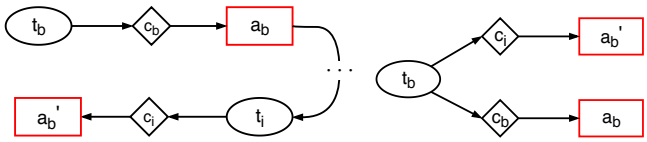
We now enumerate the space of inter-rule vulnerabilities in terms of properties of IoT information flow graphs. We will do so with respect to a benign rule $R_b = \{t_b, c_b, a_b\}$ and (when necessary) an interference rule $R_i = \{t_i, c_i, a_i\}$.

Condition Bypass. Security-sensitive actions (e.g., open the window) are often guarded by some security conditions (e.g., I am at



(a) Not enough rules (b) Active blocking

Figure 4: Condition blocking scenarios. In 4a, removing a_i will make c_b unsatisfiable. In 4b, a_i's activation makes c_b unsatisfiable.



(a) Action reverting (b) Action conflict

Figure 5: (a) Action reverting: a'_b has the opposite effect as action a_b. (b) Action conflict: t_b activates a_b and a'_b in an unknown order.

home). However, when a trigger is fired, all associated rules are activated; if there are multiple paths to the security-sensitive action, the burden is on the user to apply the condition for all active rules. The security guarantee of an action thus follows the *weakest precondition*, creating the potential for condition bypass:

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

Condition bypass is visualized in Figure 3. As an example of the condition bypass threat, consider the rule *"If temperature is higher than 30 °C, when I am at home and time is between 8am to 6pm, then open the window"*. If another rule exists with a null condition, i.e., *"If temperature is higher than 30 °C, then open the window"* then the prior condition is trivially bypassed.

Condition Block. An alternate vulnerability related to conditions is that a given condition is simply unsatisfiable. Broadly, the definition for condition blocking can be given as follows:

$$\forall\, p \in P,\ ACTIVATE(a_i) \implies OFF(c_b)$$

We identify two scenarios in which condition blocking is a potential issue, *Not Enough Rules* and *Active Blocking*, visualizations for which are shown in Figure 4. For the former scenario, a condition may depend on other devices' states but there is no rule to manipulate the state in such a way to satisfy the condition. For example, if a user has a rule *"If motion is detected at the door when home is in armed state, then send me a notification"*. If no action in the deployment sets the home's security system to the armed state, this condition cannot be satisfied. Conversely, when Active Blocking occurs there is a buggy or malicious rule that actively disables the condition before the action can be activated. For example, another rule using the "If motion is detected at the door" trigger could specify an action that sets the home's security system to the disarmed state. In either case, the user's intended action is unreachable.

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

$$\exists\, p \in P\ s.t.\ ACTIVATE(a_b) \implies ACTIVATE(a'_b)$$

Action reverting is shown in Figure 5a. The reverting action pair shown here could be `lock` and `unlock` commands on a door. It is also possible that $a_b = a'_b$, e.g., an action that toggles a switch.

Action Conflict. In contrast to action reverting, which deterministically negates a_b, action conflicts activate a_b and a'_b in a non-deterministic ordering, potentially putting the deployment in an unstable or unknown state. Action conflicts are defined as:

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

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

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

$$\exists\, p \in P\ s.t.\ ACTIVATE(a_b) \implies ACTIVATE(a_b)$$

An example of action loop are the rules *"If the bedroom light is turned on, then turn off the living-room light"* and *"If the living-room light is turned off when the home state is away, then turn on bedroom light"*. Further, attacks that exploit the action loop condition have previously been presented in the literature. For example, an attacker can use an action loop on a smart bulb to create strobe light that could potentially induce seizures [76]. An attacker can also use action looping as a side channel to leak information [49].

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

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

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

5 IRULER

In this section, we describe *i*RULER, our tool to detect inter-rule vulnerabilities in TA rulesets. The architecture and workflow of *i*RULER is shown in Figure 6. Given a set of IoT apps from a TA platform, the *Rule Parser* extracts trigger-action rules from the apps

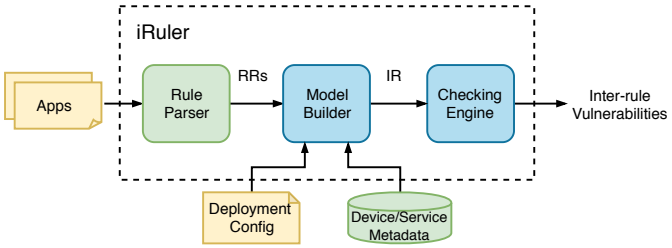


Figure 6: The architecture and workflow of *i*Ruler. RR: Rule Representations; IR: Intermediate Representation.

Listing 1: The *i*Ruler rule representation format.

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

and transforms the rules into Rule Representations (RR). The *Model Builder* takes the rule representations, device metadata and the user's deployment configuration as input and generates an Intermediate Representation (IR) of the IoT deployment. The *Checking Engine* performs checking over the IR and outputs potential inter-rule vulnerabilities as introduced in Section 4. It is then up to the user to determine the severity of the warning and whether or not to correct the rules. In Figure 6, the components in yellow are provided by the user, the components in green are platform-specific and the components in blue are platform-agnostic. Our tool can be easily extended to another platform by implementing a rule parser and building device metadata for the platform. Below we discuss each component in more detail.

5.1 Rule Parser

An IoT app could contain multiple TA rules. The rule parser first extracts all the rules in the app, then transforms the rules into uniform rule representations which are used by the model builder. Listing 1 shows the format of our rule representation. A *rule* is composed of a trigger and one or more actions. A *trigger* is defined as an event with a constraint and an *event* is defined in terms of **subject** (e.g., a certain device) and **attribute**. For example, the trigger "if temperature drops below 30" is represented as *temperature_sensor.temperature* < 30. The event here is the value change in the measurement of the temperature sensor. An *action* comprises a **condition**, the **subject**, the **command** to execute and the **arguments** to the command. A *condition* or a *constraint* could be null (i.e., no condition) or a logical expression. The difference between them is that a constraint is a predicate over the event data while a condition could be a predicate over other subjects.

5.2 Formal Modeling with Model Builder

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

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

Device State Transitions. To model the interaction of rules, it is important to model the state transitions of devices (or services) as the action of a rule could cause a state transition which invokes the trigger of another rule. For a device command that can change the device's attributes, we model the command execution as a transition from one device state to another. The value change of a device attribute is modeled as a device event. For example, the `turn_off` command of the heater is modeled as a transition from state $< switch : on >$ to state $< switch : off >$ with a switch change event $Event(oid, switch : off)$ where *oid* is the id of the device.

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

Time Modeling. We support temporal behavior modeling by modeling time as a monotonically increasing variable. Time advances when there is no other transition available. Time-based triggers (e.g.,

a timer at 8 am) are modeled as time events when the time variable advances to the specific values. For device actuation that can affect environmental variables, we make state transitions of the environment objects to update their values as time advances. The updates are made based on the effects caused by the actuation. Currently, we support *increase* (i.e., increasing by a rate), *decrease* (i.e., decreasing by a rate) and *change to* effects (i.e., directly changing to a value). For example, if a heater increases the temperature with a rate r. For each time unit that the `switch` attribute of the heater is "on", we make a state transition from $< env.temperature \mid value : T >$ to $< env.temperature \mid value : T + r >$. If no rate r is provided, we use 1 as default. One optimization we use to reduce system states is to update the values of time and environmental variables only with the values used in the ruleset. For example, if there are two timers, one at 8 am and the other at 9 am , in the rules, we will advance time from 0 to 8 am then to 9 am instead of advancing the time by one time unit in the transitions.

Device/Service Metadata. The device metadata contains the necessary information for device modeling and environment modeling. For example, it defines the attributes and commands of a device type, the effects on environmental variables (e.g., increasing temperature) of a command, and state transitions of a device command (i.e., what events will be generated by the execution of a command). Device/Service metadata can be constructed by analyzing the documentation of an IoT platform or provided by the platform developers or experts [28, 51]. For the IFTTT platform, we construct the service metadata by crawling the web page of each service to get what triggers and actions the service supports. We describe how we extract state transitions of service actions using NLP techniques in Section 6. We show examples of a device metadata and a service metadata in Appendix C.1. The service metadata is generated with the help of the NLP techniques in Section 6.

Intermediate Representation. The model builder could generate intermediate representation for different model checkers. Due to its maturity and expressiveness, we use Maude [13], which is a language and tool that supports the formal specification and analysis of concurrent systems in rewriting logic [62], as our checking engine. With rewriting logic, an IoT system, which is a concurrent system, can be naturally specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational theory describing system states, and R rewrite rules describing the system's concurrent transitions. Rewrite rules of the form crl $[l] : t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$ **if** $\phi(\overrightarrow{x}, \overrightarrow{y})$ describe an l-labeled transition in an open system from an instance of t to the corresponding instance of t'; the extra variables $\overrightarrow{y}$ on the right-hand side of the rule are fresh new variables that can represent external nondeterminism (e.g., sensor probing); ϕ is a constraint solvable by an SMT solver. In the generated intermediate representation, devices, environmental variables and time are modeled as objects; events and commands are modeled as messages; state transitions and trigger-action rules are modeled as rewrite rules (*rl* for rules and *crl* for conditional rules). Consider an example of an IoT deployment consisting of a temperature sensor `sensor` sensing the temperature from the environment and an air conditioner `ac`, which collaborate to maintain the in-house temperature at a desired setpoint. In this case, the state of the system can be modeled as $< ac \mid setpoint : \_,\ switch : \_ >, < sensor \mid temp : \_ >,$ $< env.temp \mid temp : \_ >$ and $< Time \mid time : \_ >$, where the attributes *time*, *temp*, and *setpoint* are integers representing the wall-clock, the temperature in the house, and the desired temperature setpoint, respectively, and the attribute *switch* is a Boolean referring to whether the air conditioner is turned on or off. Note that *time* and *temp* are under control of the environment, while *setpoint* and *switch* are under control of the system. The state transitions can then be modeled by the following three rewrite rules:

```
crl [turn-on] :
    (< ac | setpoint:S, switch:false >
     < sensor | temp:T > ; φ)
→   (< ac | setpoint:S, switch:true >
     < sensor | temp:T > ; φ ∧ T > S) if sat(φ ∧ T > S) .

crl [turn-off] :
    (< ac | setpoint:S, switch:true >
     < sensor | temp:T > ; φ)
→   (< ac | setpoint:S, switch:false >
     < sensor | temp:T > ; φ ∧ T ≤ S) if sat(φ ∧ T ≤ S) .

rl [time-advance] :
    < Time | time:R > < Temp | temp:T > < sensor | temp:T >
→   < Time | time:R+1 > < Temp | temp:T' > < sensor | temp:T' > .
```

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

5.3 Formal Analysis by Checking Engine

The checking engine takes the IR as input and uses *rewriting modulo SMT* [75] to discover inter-rule vulnerabilities. Rewriting modulo SMT is a symbolic technique combining the power of rewriting modulo theories, SMT solving, and model checking. For each combination of device states, we use it as an initial state to check the vulnerable properties as defined in Section 5.2. Since our goal is to find existence of violations, we use the *search* command to search a reachable state that reveals the vulnerabilities. As an example, the following search command looks up to 1 solution and a search depth 15 for a reachable state in which the air conditioner is turned on, while the temperature sensed by the sensor from the house does not exceed the current setpoint:

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

Note that the *true* on the left-hand side of the arrow indicates no initial constraints. Similar with [70], we perform bounded model checking [25, 26] with the argument like "[1,15]" to bound the search task to a certain depth to reduce the search space. The search-based model checker returns either a vulnerable state reachable from the initial state or no solution, indicating no such vulnerability.

Besides the inter-rule vulnerabilities, our tool can also check other properties using the built-in LTL (Linear Temporal Logic) [19]

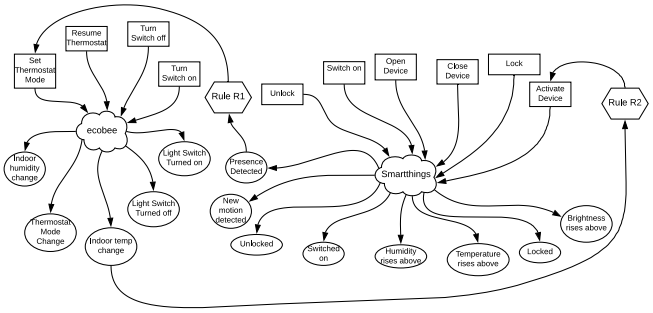


Figure 7: Initial attempts at building trigger-action information flow graphs suffered from state explosion and false dependencies.

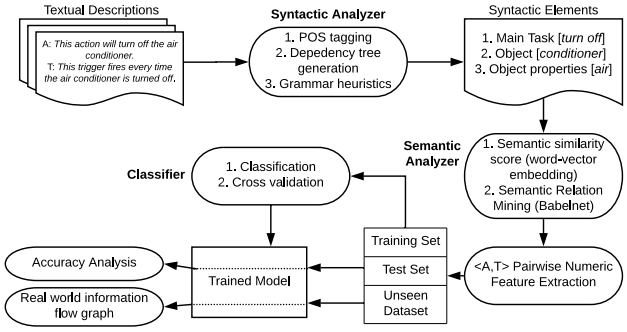


Figure 8: An overview of our NLP-based information flow analysis of trigger-action IoT platforms.

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

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

6 IOT INFORMATION FLOW MODELING

As discussed in the prior section, *i*Ruler requires an understanding of how triggers and actions interact to detect inter-rule vulnerabilities. In this section, we describe our approach to the automatic extraction of such flows from proprietary trigger-action platforms. At first glance, identifying such flows seems trivial. However, in practice, identifying these links proves surprisingly difficult.

Preliminary Experiment: Following the methodology of [86] and [63], we scraped the descriptions of 674 services and 315,393 applets from the IFTTT website. Recall that each *trigger* and *action* in an applet represent an API defined by a third-party *service* (channel). For example, the *SmartThings* service provides an action *"Lock a SmartThings device"* and a trigger *"If a SmartThings device is turned on"*. The simplest way to model action-to-trigger flows within a service is to conservatively assume that all outbound triggers depend on all inbound actions. However, applying this naïve strategy generates 6637 intra-service flows, many of which are spurious and represent false dependencies. For example, in Figure 7, the *"Lock"* action of the SmartThings would not affect the *"Humidity rises above"* trigger; these are two independent attributes that can be manipulated through this service. *Thus, while information flow within a rule (trigger-to-action) is definitionally apparent, understanding inter-rule dependencies (action-to-trigger) requires decomposition of services into their underlying components so that true flows can be identified.*

6.1 NLP-based Information Flow Analysis

Given the proprietary nature of trigger-action IoT platforms, our options for analyzing the internal state of services are extremely limited. As observed in prior work [81, 84, 86], analyzing text descriptions of IoT components that appear on the platform websites

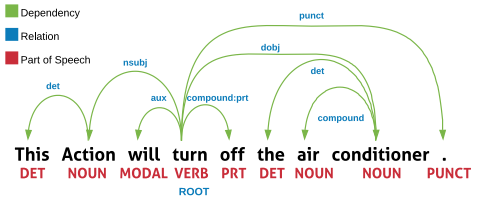


Figure 9: An example dependency tree that encodes the grammatical structure of an action description.

provides one means of overcoming this obstacle. We now present an approach that leverages Natural Language Processing (NLP) in the design of an information flow analysis framework, an overview of which is given in Figure 8. To eliminate the spurious flows and to detect the true information flows, we pose this problem as a supervised classification problem. Our framework learns a function to map an Action (A) and Trigger (T) pair from a Service (S) to a binary output specifying whether an information flow exists from A to T. As a first step toward our goal, we need to encode each $\langle A, T \rangle$ pair as a set of numeric features.

6.1.1 Syntactic Element Extraction. To simplify the analysis of unstructured text, we first perform *Part-of-Speech (POS)* tagging and *Dependency Parsing* using the Stanford CoreNLP [12] library to produce a dependency tree for the description of each rule component. An example dependency tree for an action description is shown in Figure 9. The parser performs *Dependency Parsing* to identify the *root verb* representing the main task of the rule component. All other syntactic units are either directly or indirectly connected to the root by dependency edges, which encode a grammatical relation between a source node (governor) and a destination node (dependent). While there are many dependency relationships, those we use in our analysis are:

- *Direct Object:* The dependent of this relation with respect to the root is the object that the main task is acted upon.
- *Compounds:* These relations are part of the root verb or direct object of the task (e.g., "air conditioner", "turn off").
- *Modifiers:* These relations encode words that modify the meaning of a noun by specifying some additional quality, association, or attribute (e.g., "new subscriber" - adjective modifier to subscriber).

Table 2: A summary of our feature vector, calculated as a comparison between the text descriptions of an action and trigger.

Feature	Type	Description
Verb Similarity	Continuous	Semantic similarity scores of $\langle A, T \rangle$ verb-pairs.
Object Similarity	Continuous	Semantic similarity scores of $\langle A, T \rangle$ object-pairs.
Verb Synonym	Binary	Is the trigger verb a synonym of the action verb?
Verb Hypernym	Binary	Is trigger verb (*move*) a hypernym[1] of action verb (*walk*)?
Verb Causation	Binary	Can the trigger verb (*eat*) be caused by the action verb (*feed*)?
Verb Entailment	Binary	Does the trigger verb (*wake*) entail[2] the action verb(*sleep*)?
Object Synonym	Binary	Is the trigger object a synonym of the action object?
Object Hypernym	Binary	Is trigger object (*publication*) a hypernym of action object (*book*)?
Object Meronym	Binary	Is the trigger object (*lock*) a meronym[3] of action object (*door*)?
Object Holonym	Binary	Is the trigger object (*door*) a holonym[4] of the action object (*lock*)?
Object-Property Match	Binary	Does action object property match the trigger object property?
Verb-Particle Match	Binary	Do the verb-particles[5] match between action verb and trigger verb, if the verbs are multi-word expression (*turn off*)?

Usually, *root* defines the main task while *Direct Object* and *Compound* together define the object, in addition to *Modifiers* that define the properties of the object (Figure 9). However, sometimes the clausal complement to the root verb describes the main task instead. For example, in the trigger description "This Trigger fires every time an audio event is detected", the root verb "fire" is not the main task, but "detect" is, which is a clausal complement to "fire". Moreover, "audio event" is a *passive nominal subject* to "detect" instead of the Direct Object relationship. So, there are a few other dependency relations, e.g., *Nominal* and *Passive Nominal* subjects and *Clausal Complements* [21], that we track to detect syntactic elements in order to accommodate the variability in unstructured text. These grammatical dependencies comprise the syntactic elements of interest for the remainder of our analysis.

After performing POS tagging, parsing and extracting the relevant syntactic elements, we also attempt to detect and exclude the Named Entities [9] from each text description. In preliminary experimentation, we found that this was necessary because named entities appearing in extracted object descriptions often seemed to encode similarity between dissimilar objects. For example, *WeMo Humidifier* and *WeMo Lighting* are likely to be unrelated in spite of a shared Named Entity *WeMo*. We therefore decide to exclude named entities to avoid bias when calculating object similarity.

6.1.2 Semantic Feature Extraction. After extracting the relevant text elements, we then encode the semantic relationship between the syntactic elements of the action and trigger as a vector of (continuous and binary) numerical features. These features are calculated by processing the syntactic elements of A and T in a pairwise fashion (i.e., verb-verb, object-object). Intuitively, if the elements of the trigger and action description have related semantics, it is likely that there exists a dependency between them. A summary of the feature vector is given in Table 2.

Continuous Feature Computation: We leverage the *Word Vector Embedding* technique to calculate *Verb Similarity* and *Object Similarity* features, which maps words from a vocabulary into vectors of real numbers. These vector representations are able to encode fine-grained semantic regularities using vector arithmetic [64]. Based on vector arithmetic, we then use the word embedding tools (e.g., word2vec [16], GloVe [18]) to calculate a real number score representing the semantic similarity between the two syntactic elements. We calculate pairwise *semantic similarity scores* for each pair of verbs and objects extracted from A and T. To calculate the similarity score for multi-word elements, we calculate a phrase vector as the average of the vectors of the component words [47]. Let phrase P be composed of words $(w_1, w_2, \ldots, w_n)$ with vector embeddings $(u_{w_1}, u_{w_2}, \ldots, u_{w_n})$. The vector for P is then defined as: $u_P := \frac{1}{n} \sum_{i=1}^{n} u_{w_i}$. Finally, the semantic similarity score for the action and trigger phrases is calculated as the *cosine similarity* between the two vectors.

Binary Semantic Feature Computation: We are ultimately interested in specific *causal* relationships between actions and triggers, but our continuous features reflect *any* relationship between the syntactic elements. As a result of this broader focus, the similarity scores may underweight the relationship between two elements within the context of IoT; for example, *word2vec(lock, door)* with the Wikipedia-trained model we used yields a middling similarity score of 0.53, but in the IoT domain it is highly likely that a change in lock state suggests a change in door state. To correct for this, we also calculate a series of binary features for each action trigger pair, which we define to capture generic semantic relationships that we found were commonly relevant to action-trigger flows during manual coding of our IFTTT dataset. For example, multi-word expressions (e.g., "turn on") are commonly found in descriptions, but the verb particle *on* is often tagged as a preposition by the POS tagger, so we introduce a feature that tests if the verb particles match. These features are calculated using the lexical database Babelnet [69], annotated and interlinked with semantic relations.

6.2 Classification Problem

We cast information flow detection as a supervised binary classification problem between an action and trigger pair $\langle A, T \rangle$, where both T and A belong to the same service S. Each $\langle A, T \rangle$ pair is labeled such that 1 signifies the existence of a flow from A to T while 0 signifies the contrary. We divide the dataset into training and test sets by *service* so that the classifier is unable to leverage service-specific semantics when classifying test samples. We use 4 different classification algorithms - Support Vector Machine, Random Forest, Multilayer Perceptron and Logistic Regression. We use Grid Search with Cross Validation to search the hyperparameter space to optimize the classifier performance for a high recall (i.e., maximize proportion of actual positives identified correctly) value. The decision of recall optimization comes from the intuition that it is safer to admit false flows than exclude true flows.

One issue with our dataset is that it is highly imbalanced because there are more spurious non-flows than true flows, i.e., the number of positive examples is far less than the number of negative examples. We use two different techniques to combat this problem. First, we use class-weights inversely proportional to the percentage of class examples in the training set. This assigns a higher misclassification penalty to training instances of the minority class. Second, we use *Random Oversampling* to balance the data by randomly oversampling the minority class. We do not use undersampling of majority class since we have a limited sized dataset.

[1]Hypernym: generic term used to designate a class of specific instances.
[2]Entailment: the trigger verb cannot happen unless the action verb happens.
[3]Meronym: a constituent part, the substance of, or member of some object.
[4]Holonym: The name of the object of which the meronym names a part.
[5]The verb-particles, i.e., *Off* or *On* are tagged differently by the POS-Tagger than *On* as a preposition.

Table 3: A summary of the classification performance (percentages).

Classifier	Accuracy	AUC	Recall	FP Rate	FN Rate	False Flow Reduction
SVM (RBF Kernel)	80.2	79.8	90.7	22.3	9.2	72
Random Forest	85.7	80.5	88.2	15.2	11.8	78.7
Multilayer Perceptron	86.8	82.7	88.6	16.4	11.4	77.6
Logistic Regression	83.1	79.5	84.4	20.4	15.6	74.3

6.3 Classification Performance

Based on the methodology described above, we now evaluate the overall accuracy of our NLP-aided information flow analysis tool.

6.3.1 Experimental Setup. Our feature extraction tool was implemented using the *Stanford CoreNLP* [12] library for POS tagging during syntactic element extraction, the *FastText* [27] project's Wikipedia dataset word vectors to calculate similarity scores, and Babelnet [69] to extract binary semantic features.

The classifier's training and test sets were derived by randomly selecting 512 services from our IFTTT dataset, which is described in greater detail in Section 7.1. We divided this into 374 services for training and 138 services for testing. Because we were unable to purchase, configure, and run all of the devices and services on IFTTT, our labeled ground truth is based not on applet invocations, but on manual coding by two of the authors; we consider the potential limitations of this approach in Section 8. Each coding decision entailed examining the text descriptions of the service to determine whether it was possible for a given action to lead to the invocation of a given trigger. The existence or absence of a flow was usually obvious; occasionally the coders needed to look up the functionality of a service if they were not familiar with it. Manual coding entailed an author spending approximately 40 hours manually coding the intra-service flows, followed by a second author spending 5 hours on reliability coding [82]. for a total of 45 hours of human effort. There were a small number (less than 10 in total out of 512 services) of discrepancies identified by the reliability coder, which were easily resolved between the two coders through a brief discussion. While this strategy for deriving intra-service flows is already tedious, we argue that it will shortly become entirely untenable as IoT platforms continue to grow in popularity. There is already evidence that this expansion of IFTTT is underway – during a 5 month window in 2017, the platforms services, triggers and actions grew by 11%, 31%, and 27% respectively [63].

6.3.2 Results. We used the training set to train the classifier and the test set to compute the accuracy and AUC score. Then we fed the entire training set and test set together to our tool and computed recall, error rates and the amount of false flow reduction. These results are summarized in Table 3. We compare the performance of our NLP-aided tool using different classification models against the baseline naïve strategy used in our preliminary experiments, which conservatively assumes a flow exists between all actions and triggers of a service. *Compared to this baseline which generates* 6637 *flows, our NLP-based tool with SVM classifier minimizes the FN rate to* 9.2% *while causing an overall reduction in graph complexity of* 72%. This finding demonstrates that an NLP-based approach is a first step towards overcoming the opacity of IoT platforms.

6.3.3 Discussion. In light of the large number of false dependencies that exist using the naïve information flow strategy, we feel that our error rates are promising. Here, a false positive signifies that our attack surface model is overly conservative, encoding a flow between two rules that does not actually exist, while a false negative fails to identify a legitimate flow.

We identify two error sources that can be directly attributed to our methodology. First, our approach depends on an accurate text description of the rule behavior; in cases where the trigger/action do not contain verbs that explain the behavior, we are unable to identify the flow (*True Error*). In a few cases, the classifier's decision boundary detected $\langle A, T \rangle$ pairs with high verb or object similarity as a non-flow, or pairs with lower similarity scores as a flow (*Classification Error*). However, we did not want to overfit our model to the dataset, so we restrained from fine-tuning the classifier to address this.

The larger sources of error in our system can be attributed to limitations in the underlying NLP tools we employed. (1) Text descriptions that generated complex syntax trees (with uncommon grammatical relations) led to false positives because we were unable to track the language elements indicating a non-flow (*Syntax Tree Complexity*). (2) The POS-tagger sometimes labeled words incorrectly, leading to errors; for example, the *"on"* in "Turn on" might be detected as preposition instead of a verb-particle, third-person verbs sometimes detected as plural nouns, or the word *everytime* is detected as a verb (*POS Tagger Error*). (3) Parsing errors by the CoreNLP parser module produced incorrect dependency trees, leading to incorrect feature vectors (*Dependency Parsing Error*). (4) Descriptions that contained complex object modifiers led to some false positives, e.g., "*This Action will create a regular post on your Blogger blog*" and "*This Trigger fires every time you publish a new post on your Blogger blog with a specific label*" (*Complex Object Modifier*). (5) Word embeddings often assign high similarity score to contextually similar verb pairs, for example "open-close", "activate-deactivate", thus confusing the classifier to record a false positive. (6) A significant source of error was that the word embedding models we used were not trained for the IoT domain, but a more general vocabulary (i.e., Wikipedia). This was especially problematic when novel words (e.g., "cool-mode") were encountered.

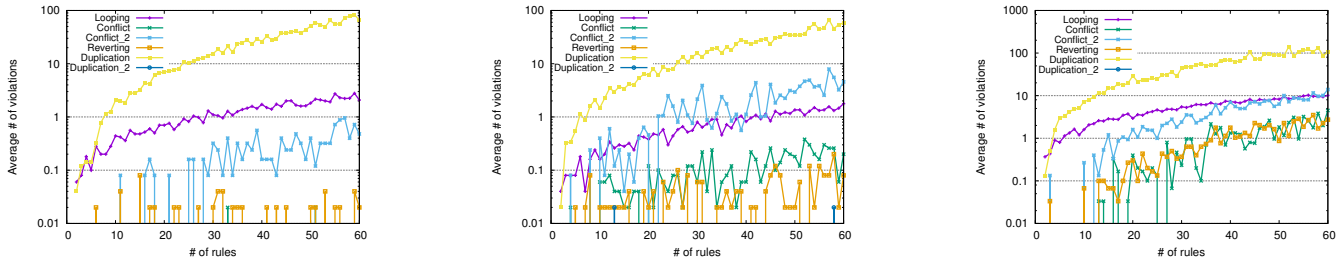
These error sources could potentially be addressed in future work through advancements in these techniques or by training NLP tools specifically for the IoT domain. Alternately, our method could be augmented with prediction uncertainty analysis and quantification techniques [43] to request human intervention when the classifier is not confident enough in its prediction.

7 EVALUATION

Having generated an information flow graph of IoT deployments using our NLP-aided analysis tool, we are now able to leverage *i*Ruler to identify inter-rule vulnerabilities within real-world IoT platforms. In this section, we examine the potential for inter-rule vulnerabilities within the IFTTT ecosystem.

7.1 Dataset

We conduct our evaluation on a dataset crawled from the IFTTT website in October 2018 using the methodology introduced by Ur et al. in [86]. The data we collect is entirely public and includes only metadata about the published applets and services – all user



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

Figure 10: Average number of vulnerabilities discovered for different configuration synthesis strategies (averaged over 50 trials per number of rules). Different inter-rule vulnerability classes are separated by color; *Conflict_2* stands for action conflict with different triggers and *Duplication_2* stands for group action duplication (i.e., one action subsumes another action). *Duplication* violations can exceed # of rules because a single action can be involved in multiple duplications.

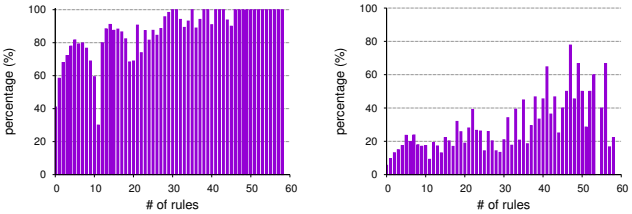
data in IFTTT is private, and thus not contained in our dataset, with the exception of aggregate applet install counts which are made public.[6] Our crawl identifies 315,393 applets and 674 services. The applets make use of 1,718 distinct triggers and 1,327 distinct actions. The applets were written by either service providers or 131,768 third-party authors (i.e., users). Some components of IFTTT applets are not publicly visible, making us unable to discover certain classes of inter-rule vulnerabilities; for example, because applet filter code is not public, we cannot analyze IFTTT for the condition bypassing vulnerability. Instead, we limit our evaluation to action loop, conflict, revert, and action duplicate vulnerabilities.

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

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

- *Install Count Strategy.* IFTTT reports the total number of installations of each applet. We normalize these install counts to assign each applet a weight and construct an IoT configuration of r rules by performing a weighted random walk starting at a random point in the IFTTT information flow graph. This strategy reflects the intuition that popular applets are more likely to be simultaneously installed.
- *Service-Based Strategy.* We construct an IoT configuration by randomly selecting a small number of services, then randomly selecting r rules from within those services. This strategy reflects the intuition that a user is likely to make use of only a small number of services.

[6]We argue that this is analogous to security surveys of mobile app markets (e.g., [37]) and therefore consistent with community norms governing ethical data collection.



(a) Including duplication (b) Excluding duplication

Figure 11: The percentage of applet authors whose applets have at least one vulnerability.

- *Author-Based Strategy.* In IFTTT, authors have the option of sharing their applets publicly. We construct an IoT configuration by assuming that an author has all of their public applets simultaneously installed. This strategy reflects the intuition that authors are likely to use their own applets.

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

7.2 Results

We apply each IoT configuration synthesis strategy for variable numbers of rules between 2 and 60, reporting the average number of discovered violations across 50 trials. Figure 10 shows the average number of vulnerabilities identified as the number of active rules increases using the Random Strategy, Install Count Strategy, and Service-Based Strategy, respectively. In Figure 10, action duplication is the most prevalent concern in the IFTTT ecosystem. Looping behaviors are also quite frequent, occurring at least once per configuration when more than 15 rules are simultaneously active. While less prevalent, we also identify the potential for conflicts and reverting behaviors in many of the synthesized configurations. The group action duplication vulnerability, while rare, was also observed in our tests. Using the Install Count Strategy, in total, 66% of the rulesets are associated with at least one inter-rule vulnerability.

We consider the Author-Based Strategy in a separate analysis because, unlike the other strategies, we are unable to control the

Table 4: Rule chaining in IFTTT. *Actions/Triggers* **is the number of chainable mechanisms in IFTTT, while** *Observed* **signifies the number of linkable mechanisms observed in at least one IFTTT rule.**

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

number of trials and the number of active rules. Figure 11a shows the percentage of authors of applets with at least one vulnerability. Almost all authors' applets show evidence of at least one inter-rule vulnerability. Again, similar to prior test, duplication is the most common concern; Figure 11b shows the frequency of vulnerabilities excluding duplication. Concerningly, about 1 in 5 authors will experience a non-duplication vulnerability in their rule set if they activate at least 10 rules. However, some authors might not simultaneously activate all their applets, meaning that this test may overestimate the frequency of vulnerabilities. However, taken as a whole, this test provides compelling evidence that inter-rule vulnerabilities currently exist in the wild.

Our study also presents an opportunity to characterize the potential for rule chaining within TA platforms. Because rule chaining increases the complexity of an IoT configuration, we theorize that it also increases the potential for security violations within the deployment. Across the 674 IFTTT services we analyzed, there exist 509 actions that can explicitly link to other rules and 518 triggers can be explicitly triggered by some action. In addition, we identify 460 actions that can affect an environment variable in order to indirectly invoke 392 triggers that monitor environmental variables. Table 4 summarizes our rule chaining results. We identify a total of 204,510 (64.8%) rules that can explicitly link to other rules, and 62,013 (19.5%) rules that can be explicitly linked by other rules. There exist 10,128 (3.2%) rules can implicitly link to other rules, and 6262 (2.0%) rules that can be implicitly linked by other rules.

7.3 Vulnerability Analysis

Condition Bypassing & Condition Blocking. While we introduce the notion of condition-based vulnerabilities in §4, we are unable to detect them on IFTTT because applets' filter code is not public. We verified the presence of condition vulnerabilities using our own applets but leave large-scale validation of this issue to future work.

Action Reverting. Our dataset contains 1127 applets with multiple actions, 50 of which contain contrary action pairs that revert each other. A rule susceptible to action-reverting by another rule/applet, usually occur within distance 1 or 2 of one another in the IFTTT information flow graph, but the longest distance observed was 5 in a configuration of 26 applets; such violations would likely to be difficult to identify manually. One example of such violation in our dataset consists of an applet that turns the lights on when motion is detected, but another applet turns off the lights whenever a light is turned on. A more concerning violation we observed was a rule that would disconnect a *HomeSeer* device from Wi-Fi the moment it was turned on, creating a DoS attack because the device cannot function or receive commands without a network connection.

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

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

Action Duplication. As seen from Section 7.2, action duplication is very common. It is perhaps not surprising to observe redundant rules in the community-based IFTTT ecosystem as developers may publish applets with the same function. A chain length of 8 in a configuration of 38 rules is the longest we observed to contain an action duplicate violation. The number of group duplication violations we detected is very small as there are only 113 applets that use group actions. We further investigated that IoT services in IFTTT provides more group actions, such as *Turn off device* vs. *Turn off all devices* (Linn) or *Disarm all cameras* vs. *Disarm a camera* (Eagle Eye NuboCam). We envision that as more functionalities are introduced in IoT devices, these superseding relationships will become more common, creating the potential for action-duplication vulnerabilities to significantly frustrate the debugging of IoT deployments.

8 DISCUSSION & LIMITATIONS

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

The IFTTT Applets Dataset. Similar to Ur's IFTTT recipe dataset in [86], our dataset is missing relevant information that is not publicly available, including values for the trigger fields in each applet and the applet's filter code (i.e., *conditions*). An interesting direction for further study is leveraging applet descriptions to attempt to recover these fields; for example, the applet *"Get a phone call alert when a door is opened during sleeping hours,"* suggests the condition "during sleeping hours" is applied to the `call_my_phone` action. Note the model checker of *i*Ruler already supports conditions.

Synthetic IoT configurations. Because we lack real-world examples of IoT deployment configurations, in our evaluation, we use heuristic strategies to synthesize IoT deployments from our IFTTT

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

Manual Coding of Action-Trigger Flows. Due to the difficulty and cost of registering for hundreds of IFTTT services, many of which would require the purchase of one to dozens of devices in order to exercise, we had to rely on manual coding (not physical invocation) as our ground truth for information flow on IFTTT. It is difficult to judge the correctness of our manual labeling without physical ground truth; however, because services are incentivized to write informative text descriptions of their functionalities, we believe that our coding was accurate enough to demonstrate the validity of our NLP approach. Regardless, this coding is a potential source of error in our analysis.

Applicability. We ensure the generality of our approach through presenting a realistic trigger-action rule model. While we have implemented *i*Ruler for IFTTT, this model holds for other systems (e.g., Zapier and Microsoft Flow), as does the observation that NLP-based analysis is required due to the closed nature of these platforms.

9 RELATED WORK

IoT Security. Numerous vulnerabilities have been identified in IoT devices [3, 45, 78], protocols [2, 42], apps and platforms [38]. Alrawi et al. [22] proposed a modeling methodology for IoT devices, associated apps and communication protocols to analyze device-specific security postures. Different from the network-based [79, 89], platform-based [39] and app-based [49, 87] IoT-security solutions which detect vulnerabilities at runtime, *i*Ruler leverages NLP and model checking to statically check vulnerabilities before an app is installed and executed. Celik et al. [29] use static analysis to identify sensitive data flows in IoT apps, while our work studies vulnerabilities caused by the interaction of *multiple* trigger-action rules. Several other studies consider challenges related to access control in IoT [44, 50, 74, 77].

Trigger-Action Programming (TAP) in IoT. Researchers have studied how smart homes [33, 85, 88] and commercial buildings [66] can be customized using TAP, and the usability of existing TAP frameworks to propose guidelines for developing more user-friendly interfaces [46, 83]. Ur et al. [86] create a dataset of IFTTT recipes and analyze different aspects of the recipes. Bastys et al. [23, 24] discuss user privacy issues in IFTTT and developed a framework to detect private data leakage to attacker controlled URLs. However, they concentrate only on the privacy violations in the filter code of individual applets, not the interaction between applets. Fernandes et al. [40] consider the effect of OAuth-related overprivilege issues on the IFTTT platform and proposed a way to decouple the untrusted cloud from trusted clients on the user's personal devices.

NLP-aided Flow Analysis. FlowCog [71] extracts app-semantics and contextual information that defines an android app behavior, and uses NLP to correlate the app behavior with the information flows in the app. Other work has used NLP to locate sensitive information in mobile apps and track information leakage to third-party libraries [35, 67]; evaluate the semantic gap between mobile app descriptions and app permissions [72], and match IoT app description with actual app behavior [84]. Ding et al. [36] use keyword identification in the app description of SmartThings apps to detect app interaction-chains through physical channels. Surbatovich et al. [81] define an information-flow lattice to analyze potential secrecy or integrity violations in IFTTT recipes. While their work manually rewrites and labels triggers and actions to identify rule-interactions chains, our approach uses NLP to automate this process. There are also efforts to build semantic parsers that creates executable code from IFTTT-style natural language recipes [53, 73] , which are orthogonal to our contributions.

IoT Automation Errors. IoT automation errors have been studied from various aspects, including analyzing logic inconsistencies and supporting end-user debugging to resolve them [20, 34, 51, 52, 90] as well as assisting IoT app developers with GDPR [58]. Chandrakana et al. [68] identify that too few triggers in automation rules is a source of errors and security issues. They propose a tool to determine a necessary and sufficient set of triggers based on the actions written by end users. However, their tool analyzes each rule in isolation while we consider vulnerabilities from rule interactions. Some work has also been done on detecting and resolving automation conflicts in smart home and office environments [59, 60, 65, 66, 80]; in this work, we consider a broader class of vulnerabilities.

IoT Properties Checking. Several recent studies have proposed to check security or safety properties of IoT when multiple rules/apps are enabled. We compare our approach with other existing approaches in different aspects in Table 5; *i*Ruler is among the works that support the more advanced features of TA platforms (*Multiple Actions*), incorporates a broad set of characteristics into its model (*Environment Modeling*, *Device Location*, *Time Modeling*, *Support Checking Other Properties*), and identifies new classes of inter-rule vulnerabilities.[7] Conversely, these works also provide several useful properties that we did not consider in *i*Ruler. AutoTap [90] presents a method for verifying configuration properties as expressed by novice users, and joins MenShen [28], Salus [51], and SIFT [52] in supporting automated creation and repair of rules (*Rule Writing*). Systems like Soteria [30], IoTSan [70], and HomeGuard (arXiv preprint only: [32]) are based on source code analysis of IoT apps and can therefore consider additional factors such as finer-grained reduction of state explosion and specific malicious input sequences. IoTGuard [31] instruments apps to check security and safety properties at runtime. Conversely, rather than leverage source code analysis, instrumentation, or a priori knowledge of app behaviors, our technique uses an NLP-based approach to infer information flow. As a result, *i*Ruler is necessarily less precise and fine-grained in its analysis but has the advantage of working out-of-the-box on commodity IoT platforms where source code is typically unavailable.

[7] We show the vulnerabilities considered by other work in Table 6 in Appendix B.

Table 5: Approaches for checking security and safety properties of IoT rules/apps. [31] checks properties at runtime while all others perform static checking. These systems all have different aims and advantages; this table focuses specifically on the similarity of their design to *i*Ruler.

	Multiple Actions	Environment Modeling	Device Location	Time Modeling	Support Checking Other Properties	# Inter-rule Vuln Types Considered	Rule Writing	Access Required
*i*Ruler	✓	✓	✓	✓	✓	**8**	✗	Applet Descriptions, Config Data
Soteria [30]	✓	✗	✗	✗	✓	3	✗	App Source Code[1], Device Handler Code
IoTSan [70]	✓	✗	✗	✓	✓	2	✗	App Source Code, Config Data
AutoTap [90]	✗	✗	✗	✓	✓	N/A	✓	App Behaviors,[2] Device Specifications
MenShen [28]	✗	✓	✗	✓	✓	N/A	✓	App Behaviors,[2] Device Schema
Salus [51]	✗	✓	✓	✓	✓	N/A	✓	App Behaviors,[2] Config Data, Device Schema
SIFT [52]	✗	✓	✗	✓	✓	1	✓	App Behaviors,[2] Device Metadata
HomeGuard [32]	✓	✗	✗	✗	✗	5	✗	App Source Code, Config Data
IoTGuard [31]	✓	N/A	✗	N/A	✓	3	✗	App Source Code, Instrumentation
Surbatovich et al. [81]	✗	✗	✗	✗	✗	N/A	✗	Applet Descriptions[1]

1. Configuration factors are not considered.
2. Assumes knowledge of app behaviors is available a priori.

10 CONCLUSION

While the trigger-action programming paradigm promotes the creation of rich and collaborative IoT applications, it also introduces potential security and safety threats if users do not take precautions in combining these apps. In this work, we generalize and examine inter-rule vulnerabilities in trigger-action IoT platforms, presenting a tool for their automatic detection. *i*Ruler combines the power of SMT solving and model checking to model the IoT systems and check vulnerable properties. As a related contribution, we have also demonstrated an NLP-aided technique for inferring information flow between rules in proprietary trigger-action platforms.

ACKNOWLEDGEMENTS

This work was supported in part by NSF CNS 13-30491, NSF CNS 17-50024, and NSF CNS 16-57534. The views expressed are those of the authors only. We appreciated valuable insights from our CCS reviewers and our shepherd, Blase Ur.

REFERENCES

[1] 2014. Gartner Says the Internet of Things Will Transform the Data Center. http://www.gartner.com/newsroom/id/2684616.
[2] 2015. Critical Flaw identified In ZigBee Smart Home Devices. https://goo.gl/BFBa1X.
[3] 2016. Mirai Attacks. https://goo.gl/QVv89r.
[4] 2018. Apple HomeKit. http://www.apple.com/ios/home.
[5] 2018. IFTTT. https://ifttt.com.
[6] 2018. IFTTT Home Security Applets. https://ifttt.com/search/query/home%20security.
[7] 2018. Iris by Lowe's. https://www.irisbylowes.com/.
[8] 2018. Microsoft Flow. https://flow.microsoft.com.
[9] 2018. Named Entity Recognition (NER) and Information Extraction (IE). https://nlp.stanford.edu/ner/.
[10] 2018. openHAB. https://www.openhab.org/.
[11] 2018. SmartThings. https://www.smartthings.com.
[12] 2018. Stanford CoreNLP. https://stanfordnlp.github.io/CoreNLP.
[13] 2018. The Maude System. http://maude.cs.illinois.edu/w/index.php?title=The_Maude_System.
[14] 2018. TypeScript. https://www.typescriptlang.org/.
[15] 2018. Wink. https://www.wink.com/.
[16] 2018. Word2Vec. https://code.google.com/p/word2vec.
[17] 2018. zapier. https://zapier.com/.
[18] 2019. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove.
[19] 2019. Linear temporal logic. https://en.wikipedia.org/wiki/Linear_temporal_logic.
[20] 2019. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies* 123 (2019), 56 – 69.
[21] 2019. Universal Grammatical Dependency Relation Definitions. http://universaldependencies.org/u/dep/.
[22] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 208–226.
[23] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What?: Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1102–1119.
[24] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. 2018. Tracking Information Flow via Delayed Output. In *Secure IT Systems*, Nils Gruschka (Ed.). Springer International Publishing, Cham, 19–37.
[25] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 193–207.
[26] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. 2003. Bounded model checking. *Advances in computers* 58, 11 (2003), 117–148.
[27] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* (2016).
[28] Lei Bu, Wen Xiong, Mike Chieh-Jan Liang, Shi Han, Shan Lin, Dongmei Zhang, and Xuandong Li. 2018. Systematically Ensuring The Confidence of Real Time Home Automation IoT Systems. *TCPS (ACM Transactions on Cyber-Physical Systems)* (June 2018).
[29] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1687–1704.
[30] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 147–158.
[31] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*.
[32] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. *CoRR* abs/1808.02125 (2018).
[33] Luigi De Russis and Fulvio Corno. 2015. HomeRules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2109–2114.
[34] Luigi De Russis and Alberto Monge Roffarello. 2018. A Debugging Approach for Trigger-Action Programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18)*. ACM, New York, NY, USA, Article LBW105, 6 pages.
[35] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.
[36] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 832–846.
[37] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.
[38] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE S&P*.

[39] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*.

[40] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2017. Decoupled-IFTTT: Constraining Privilege in Trigger-Action Platforms for the Internet of Things. *arXiv preprint arXiv:1707.00405* (2017).

[41] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*.

[42] Behrang Fouladi and Sahand Ghanoun. 2013. Honey, I'm home!!-Hacking Z-Wave Home Automation Systems. *Black Hat USA* (2013).

[43] Z Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521, 7553 (May 2015), 452–459.

[44] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 255–272.

[45] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *ASIA CCS*.

[46] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Ubicomp*. 215–225.

[47] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*.

[48] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 21.

[49] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*.

[50] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. 2017. FACT: Functionality-centric Access Control System for IoT Programming Frameworks. In *Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies (SACMAT '17 Abstracts)*. ACM, New York, NY, USA, 43–54.

[51] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys '16)*. ACM, New York, NY, USA.

[52] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. ACM, 298–309.

[53] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*. 4574–4582.

[54] Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. 2018. ROLA: A New Distributed Transaction Protocol and Its Formal Analysis.. In *FASE*. 77–93.

[55] Si Liu, Peter C Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. 2018. *Read atomic transactions with prevention of lost updates: ROLA and its formal analysis*. Technical Report.

[56] Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer. 2018. Formal modeling and analysis of the Walter transactional data store. In *International Workshop on Rewriting Logic and its Applications*. Springer, 136–152.

[57] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic analysis of consistency properties of distributed transaction systems in Maude. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 40–57.

[58] Tom Lodge, Andy Crabtree, and Anthony Brown. 2018. IoT App Development: Supporting Data Protection by Design and Default. In *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers (UbiComp '18)*. ACM, New York, NY, USA, 901–910.

[59] Hong Luo, Ruosi Wang, and Xinming Li. 2013. A rule verification and resolution framework in smart building system. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 438–439.

[60] Meiyi Ma, S Masud Preum, W Tarneberg, Moshin Ahmed, Matthew Ruiters, and John Stankovic. 2016. Detection of runtime conflicts among services in smart cities. In *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*. IEEE, 1–10.

[61] José Meseguer. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science* 96, 1 (1992), 73–155.

[62] José Meseguer. 2000. Rewriting logic and Maude: Concepts and applications. In *International Conference on Rewriting Techniques and Applications*. Springer, 1–26.

[63] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, New York, NY, USA, 398–404.

[64] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*.

[65] Sirajum Munir and John A Stankovic. 2014. DepSys: Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*. IEEE, 127–138.

[66] Alessandro A Nacci, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. 2015. Buildingrules: a trigger-action based system to manage complex commercial buildings. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. ACM, 381–384.

[67] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[68] Chandrakana Nandi and Michael D Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *PLAS*. 97–102.

[69] Roberto Navigli and Simone Paolo Ponzetto. 2012. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence* 193 (2012), 217 – 250.

[70] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 191–203.

[71] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. 2018. FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1669–1685.

[72] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 527–542.

[73] Chris Quirk, Raymond J Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes.. In *ACL (1)*. 878–888.

[74] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash. 2018. Tyche: A Risk-Based Permission Model for Smart Homes. In *2018 IEEE Cybersecurity Development (SecDev)*, Vol. 00. 29–36.

[75] Camilo Rocha, José Meseguer, and César Muñoz. 2017. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming* 86, 1 (2017), 269–297.

[76] Eyal Ronen and Adi Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *EuroS&P*. 3–12.

[77] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational Access Control in the Internet of Things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1056–1073.

[78] Vijay Sivaraman, Dominic Chan, Dylan Earl, and Roksana Boreli. 2016. Smart-Phones Attacking Smart-Homes. In *WiSec*. 195–200.

[79] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. 2015. Network-level security and privacy control for smart-home IoT devices. In *WiMob*. 163–167.

[80] Yan Sun, Xukai Wang, Hong Luo, and Xiangyang Li. 2015. Conflict detection scheme based on formal rule model for smart building systems. *IEEE Transactions on Human-Machine Systems* 45, 2 (2015), 215–227.

[81] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web*. 1501–1510.

[82] Moin Syed and Sarah C. Nelson. 2015. Guidelines for Establishing Reliability When Coding Narrative Data. *Emerging Adulthood* 3, 6 (2015), 375–387.

[83] Kazuki Tada, Shin Takahashi, and Buntarou Shizuki. 2016. Smart home cards: Tangible programming with paper cards. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, 381–384.

[84] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, Xiaofeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. (2017).

[85] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *CHI*.

[86] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the*

2016 CHI Conference on Human Factors in Computing Systems. ACM, 3227–3231.
[87] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *Network and Distributed Systems Symposium*.
[88] Jong-bum Woo and Youn-kyung Lim. 2015. User experience in do-it-yourself-style smart homes. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*. ACM, 779–790.
[89] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *HotNets*.
[90] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 281–291.

A IFTTT APPLET

A.1 Filter Code

In Listing 2, we show an example snippet of filter code. The code snippet conditionally execute actions based on the time of a day.

Listing 2: An example snippet of IFTTT applet filter code.

```
var timeOfDay = Meta.currentUserTime.hour()

if (timeOfDay >= 22 || timeOfDay < 8 ) {
  // Skip sending me a push notification
  IfNotifications.sendNotification.skip("Too late")
} else {
  // Skip saving the article to Feedly
  Feedly.createNewEntryFeedly.skip("I already know")
}
```

B INTER-RULE VULNERABILITIES

In Table 6, we show the inter-rule vulnerabilities considered by existing work.

Table 6: The types of inter-rule vulnerabilities considered by existing work.

	Vulnerabilities Considered
iRuler	conflict, loop, revert, duplicate, group duplicate condition bypass, action blocking, not enough rules
Soteria [30]	conflict, duplicate, inconsistent events
IoTSan [70]	conflict, duplicate
SIFT [52]	conflict
HomeGuard [32]	conflict, duplicate, loop, condition disabling, condition enabling
IoTGuard [31]	conflict, duplicate, loop

C IOT SYSTEM MODELING

C.1 Device/Service Metadata

In Listing 3, we show the device metadata of a simple heater with a `switch` attribute and two commands `turn_on` and `turn_off`.

Listing 3: An example device metadata of a simple heater.

```
{
  "ModelType":"SimpleHeater",
  "Attributes":[
    {
      "Name":"switch",
      "Type":"bool",
      "Default":"false"
    }
  ],
  "Commands":[
    {
      "Name":"turn_on",
      "Arguments":[],
      "Transition":{
        "assignments":{
          "switch":"true"
        }
      },
      "Effects":[
        {
          "EnvironmentalVariable":"Temperature",
          "Effect":"Increase",
          "Rate":1
        }
      ]
    },
    {
      "Name":"turn_off",
      "Arguments":[],
      "Transition":{
        "assignments":{
          "switch":"false"
        }
      }
    }
  ]
}
```

In Listing 4, we show the generated service metadata of the Lockitron [8] service in IFTTT with our NLP-aided information flow analysis tool. The Lockitron service has two triggers "*Lockitron locked*" and "*Lockitron unlocked*", and two actions "*Lock Lockitron*" and "*Unlock Lockitron*".

Listing 4: The service metadata of Lockitron generated with the help of our NLP tool.

```
{
  "ServiceType":"Lockitron",
  "Attributes":[],
  "Commands":[
    {
      "Name":"Lock_Lockitron",
      "Arguments":[
        {
          "Name":"lock_id",
          "Type":"string"
        }
      ],
      "Transition":{
        "Events":[
          {
            "Name":"Lockitron_Locked"
          }
        ]
      }
    },
    {
      "Name":"Unlock_Lockitron",
      "Arguments":[],
      "Transition":{
        "Events":[
          {
            "Name":"Lockitron_Unlocked"
          }
        ]
      }
    }
  ]
}
```

[8] https://ifttt.com/Lockitron