THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AUTOMATED DETECTION OF HARMFUL INTERNET OF THINGS INTERACTIONS THROUGH NATURAL LANGUAGE

JONATHAN BEES
SPRING 2020

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Patrick Drew McDaniel
William L. Weiss Professor of Information and Communications Technology in the School of Computer Science and Engineering
Thesis Supervisor

Jesse L. Barlow
Professor of Computer Science and Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Modern "smart" devices, which take everyday objects like locks and refrigerators and make them visible and controllable via the internet, make up what is referred to as the Internet of Things (IoT). The breadth of "things" which could be connected to the internet has encouraged the rapid development and release of new IoT products by many disparate companies. Because of the enormous quantity and variety of internet-connected sensors and appliances, the causes and effects of the interactions between these devices are difficult to detect. This issue arises from both the sheer number of internet-connected devices which could interact with each other as well as the fact that many of these interactions can take place through the physical environment, separate from any explicitly programmed logic. This can lead to devices interacting in ways that users did not anticipate or intend. In this thesis, we use the natural language descriptions created by device manufacturers to describe the properties their devices can sense and the actions they can perform to determine how a connected set of IoT devices could interact both directly and via the physical environment they inhabit. Using Natural Language Processing (NLP), we determine what aspects of the environment or the state of the IoT system could trigger a program, along with what changes that program's action would cause to the system state and the environment. We then use these representations to create a representation of the environmental and system state to determine what combinations of programs could exhibit undesirable properties such as compromising a user's safety. We evaluate this approach on a corpus of 280,000 trigger-action programs (17,000 unique pairs of 1,470 triggers and 896 actions) from the IFTTT platform, finding more than 350,000 possible instances of program combinations that could unintentionally exhibit undesirable properties. This approach

can be applied to the current, fractured state of the IoT ecosystem without additional work from manufacturers, third-party integrators, or users, allowing users to detect and avoid situations that could endanger them or their belongings.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I am incredibly grateful to a number of people for their help and guidance throughout this project. First, to Dr. Patrick McDaniel, for patiently guiding me through the process of performing a large-scale research project and effectively communicating complex ideas. To Dr. Berkay Celik, for providing the inspiration for this project and early guidance on its scope and implementation. To Ryan Sheatsley, for his constant encouragement and helpful input, and to the many other members of the SIIS lab who provided advice, review, and support throughout the past year.

Thesis Statement

Natural Language Processing can be used to extract meaningful state information from textual descriptions of IoT automation specifications, and that state information allows for accurate prediction of the emergence of unsafe or undesired behavior in an IoT system before it occurs.

Chapter 1

Introduction

Modern “smart” devices, which take everyday objects like locks and refrigerators and make them visible and controllable via the internet, make up what is referred to as the Internet of Things (IoT). IoT allows users not only to control the world around them using their phones, but it also gives them the ability to automate aspects of their everyday lives like unlocking their front door when they come home or opening their windows when it cools off after a hot summer day.

The breadth of “things” which could be connected to the internet has encouraged the rapid development and release of new IoT products from existing manufacturers of refrigerators, light bulbs, doorbells, blinds, and sprinkler systems, as well as startups aimed at being the first to introduce a product in an unclaimed niche. In order to make these devices work in a coordinated fashion, third-party integrating services like IFTTT and HomeKit have introduced a means of centralizing control and automation of these devices. These services work on the basis of what is known as “trigger-action programming”. When an event occurs that is detected by an IoT device (or in an online service) a message is sent to the integrating service: a “trigger” is fired. When that occurs, the integrating service sends a command to another device or service, causing it to perform an “action”.

These devices are used in homes and businesses where they are expected to fill critical roles. Some devices serve to sense characteristics of the space they occupy, like temperature, motion,

or moisture. Others are intended to perform actions in it, such as turning appliances on and off, controlling door locks, or changing the temperature by controlling a furnace or air conditioner. Because they occupy a shared physical space, these actuating devices each have effects on that space. These interactions may be difficult to predict, especially as the number of devices in a system grows. While some interactions may be benign, many can cause the system to behave in ways which are unexpected and even harmful.

1.1 Motivating Example

We define a "harmful interaction" as one in which a user's intentions are violated or the IoT system reaches a state that compromises the user's safety a due to an interaction between two programmed IoT automations. For example, the following harmful interaction could occur in a situation where an environment contains a smoke alarm, sprinkler system, moisture detector, and water shutoff valve. (Figure 1.1). In this scenario, a user has two applications enabled at the same time. One of the applications ensures that their basement doesn't flood by turning off the water supply if it detects moisture near the water supply valve, and the other suppresses fires by turning on the sprinklers if the smoke detector goes off. In isolation, each of these applications performs a task designed to prevent water and fire damage. However, if there were to be an actual fire, the sprinkler system would turn on at first, but its water supply would be shut off as soon as water reached the moisture sensor. As a result, the fire would spread unchecked and cause much more damage than it would have otherwise.

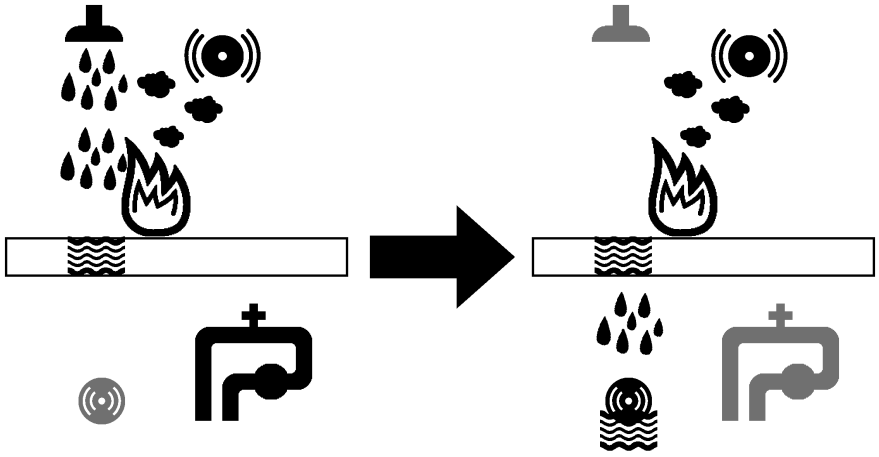


Figure 1.1: Sprinkler and Moisture Sensor Interaction

As this example shows, these harmful interactions can have real consequences for users of IoT devices. However, it is not obvious how a given set of IoT devices will actually interact. What is needed is a means of determining the aspects of a physical environment which can facilitate interaction so that later analysis can use this information to evaluate the risks device interaction through those environmental aspects might pose.

Solving this problem is made particularly complex by the fact that the individual devices in an environment often come from different manufacturers and don't have any visibility into the existence of other devices. The only party in the modern IoT ecosystem which does have some insight into how all the devices in a system work together is the third-party integration platform. Unfortunately, it is also the party with the least knowledge of and access to the devices themselves. The only interface they have to the devices is through manufacturer-provided application programming interfaces (APIs). The most reliable information that the integration platform provider and its users have about the device's functionality is through the natural language description provided by a manufacturer describing the functionality of a "trigger" or "action".

Figure 1.1 provides an example of a manufacturer-created trigger and action from the IFTTT platform. It is presented to the user as a "channel" (a grouping created by a manufacturer for their devices), a title, and a description.

	Channel	Title	Description
Trigger	Withings Home	Noise has been detected	This Trigger fires every time your Withings Home detects noise
Action	D-Link Siren	Play sound	This action will play the sound you selected on your siren.

Table 1.1: Manufacturer-provided Trigger and Action Information

1.2 Approach

In order to make use of this written information, we can use Natural Language Processing (NLP). The goal of NLP is to take unstructured human-created text and extract meaningful structured information. This information might be general in nature, such as whether the sentiment of a review is generally positive or negative, or it could be more specific, like a representation of the entities a piece of text describes and what action one entity is performing on another. In order to understand the interaction of IoT devices in an environment, the information we need to extract is:

1. What is the device?

2. What is it doing or detecting?

3. How can it affect (or be affected by) the environment in which it operates?

To answer these questions, we use NLP to identify the grammatical dependencies between words in a sentence. We identify the action the sentence describes and connect it to the object which is performing that action or having the action performed on it. To figure out whether devices might interact through changes in the state of their shared environment, we determine the environmental factors that are conceptually related to the object (e.g. what is an object typically used for?)

Once we understand what in an environment the actions can affect and what the triggers can be affected by, we need to determine how these devices could interact with each other.

We say that an interaction has occurred if the action of one program activates the trigger of another. In order to determine the combinations of devices and applications which could potentially

lead to harmful interactions, we create a representation of the state of an IoT environment, trigger a program, and see what "chains" of one program activating another emerge. We define an example set of safety properties and check those "chains" against the properties. If they are violated by that simulated environment, we infer that a potentially harmful interaction could occur with that combination of programs and devices in a real-world environment.

In this thesis we explore how we can begin to identify potential surfaces for interaction between Internet of Things devices in an automated way that is suited to the state of the modern IoT ecosystem. We identify an approach which can detect real, actionable interactions without requiring significant system redesigns or effort from platforms or manufacturers. This NLP-based approach provides a foundation for further work in performing the sorts of analyses which are necessary to ensure users can trust IoT systems to perform safely and reliably into the future.

This approach can be used to check all of the trigger-action programs on a platform so that, before a user of an integration platform enables a new program, their set of programs is compared to the sets of programs which could cause harmful interactions. The user can be informed of how the interaction may occur, but may be permitted to enable the recipe anyway if they are willing to deal with the consequences or are confident that their system is physically configured in such a way that the interaction would not take place.

Chapter 2

Background

2.1 IoT Platforms and Integration

Manufacturers of IoT devices like Philips[1], GE[2], and Arlo[3] each provide users with their own proprietary app, which is necessary to fully control and configure their brand of devices. However, these vendors recognize that users will likely have devices from other manufacturers, and that their users will likely prioritize the ability to have their devices work together to create an integrated "smart home". Instead of developing their own infrastructure to interface with all of the other vendors, those companies usually create integrations with third-party platforms like IFTTT[4], SmartThings[5], or HomeKit[6], which allow users to access and automate a subset of the device's features in concert with the rest of their IoT devices. In order to make automation capabilities accessible to general consumers, these platforms generally do not require users to interact with code. In fact, SmartThings, which previously supported user-created programs written in the Groovy programming language, is now moving users to a new version of the platform with a smaller number of curated applications, encouraging users to migrate to the same form of "trigger-action programming" presented by other platforms instead. [7]

2.2 Trigger-Action Programming

IFTTT[4] is one of the most popular services allowing users to integrate their devices through "trigger-action programming". These simple programs, referred to as "recipes" on the IFTTT platform, connect a "trigger", which can be a change in the environment detected by an IoT sensor or an event on a web service, to an "action", which can cause a change of state in the IoT system (e.g. changing the mode from "home" to "away"), actuate a physical IoT device (e.g. lock a door), or cause an event to take place on a web service (e.g. making a social media post).

Manufacturers of IoT devices and developers of web applications enable connectivity by creating "services" on the IFTTT platform (also referred to as "channels"), which consist of triggers and actions that can be combined by users to create and share recipes. These triggers and actions provide end users with the ability to interact with a manufacturer-created API through a code-free interface. For consistency, we will use the IFTTT terminology throughout this paper.

These recipes are created primarily by users of the IFTTT platform. While IFTTT does have some curated recipe categories (like gardening, smart plugs, and smart lighting), most recipes are found by browsing to a "service" category which lists all of the recipes which have a trigger or action involving a particular IoT device manufacturer or web service.

For example, if a user were to purchase a Philips Hue light bulb, they could go to the Philips Hue service category to see all of the user-created recipes related to Hue bulbs. They are then presented with a list of user-created recipe titles which include their device as either a trigger or an action. If they find a recipe they want to use, they are presented with the trigger device manufacturer's trigger title and description along with any fields the trigger has (e.g. the temperature reading from a smart thermostat at which the trigger should fire), then the action device manufacturer's action title and description along with any fields attached to that action (e.g. a specific lightbulb to turn on).

If the user does not find a recipe with the functionality they want, they can create a new recipe. To do this, they select a device manufacturer's trigger title, fill out any fields present, then select a

	Channel	Title	Description	Fields
Trigger	Ring	New Motion Detected	This Trigger fires every time a motion is detected at the given doorbell.	Which device?
Action	Philips Hue	Turn on lights	This Action will turn on your hue lights.	Which lights?

Table 2.1: Example IFTTT Recipe

device manufacturer's action title and fill out any fields present there, give their new recipe a title and description, then enable it and optionally publish their new recipe to other users on the IFTTT platform.

One recipe on the IFTTT platform which connects IoT devices from two separate manufacturers is titled "Turn on lights when there is motion at my door", and has the description "This is a good security measure in case people show up at your door unannounced. This will automatically turn on lights to make it seem as though you're home." It connects a device from Ring platform to one or more devices from Philips Hue, and a user could find it on the Service page for either device. The information visible to a user adding the recipe can be seen in Table 2.1. In this case, the recipe title and description seem to match up well with the trigger and action information provided by the manufacturers of those devices. If a "New Motion Detected" event occurs on a user's Ring doorbell, Ring's servers will communicate with IFTTT to inform it of the trigger. If a user has any recipes enabled which have that event as their trigger, IFTTT will then perform the associated action. In this case, that action is to communicate with the Philips Hue servers, sending a "Turn on lights" event with whichever lightbulbs the user specified in the recipe.

2.3 Natural Language Processing

2.3.1 Introduction to NLP

Natural Language Processing, or NLP, is the process of using algorithms to interpret the meaning of language written or spoken by humans.

Early research efforts from the 1950s to 1980s led to the development of hand-crafted grammatical structures which computers could use to interpret sentences, but this led to huge numbers of rules and poor handling of less common phrasing. However, in the 1980s, there was a move to statistical analysis of huge datasets enabled by advances in computing, and most recently to approaches based on machine learning, which allows grammatical rules and structures to be inferred directly from the dataset rather than requiring each to be hand-coded.

While NLP has come a long way since its inception, it is still an active area of research. It remains extremely difficult, for example, to determine the structure of sentences whose interpretation depends on an understanding of the real-world qualities of an item, or to create new meaningful text from raw information for humans to read. However, we assert that the state of the art has advanced to the point where it is feasible to gain meaningful knowledge about the physical world from descriptions written by humans.

2.3.2 NLP Techniques

Two important techniques for extracting information about the syntax of natural language are "part-of-speech tagging" and "dependency parsing". Part-of-speech tagging attaches a part of speech, like "noun" or "verb", to every word in a sentence. This is more complicated than just attaching a single part of speech to each word in the dictionary, because many words which are spelled the same can have different meanings in different contexts. For example, "saw" can be the past-tense of the verb "see" or can be a noun referring to an object used for cutting things. "Parsing" refers to the act of performing grammatical analysis to determine the role each word plays in a sentence. Dependency parsing is a type of parsing which focuses on the relationships between words.

To demonstrate the use of these techniques, we will use the sentence "She ran around the lake.", visualized in Figure 2.1. Below each word in this sentence, we see the part of speech. "She" is labeled a pronoun, "ran" is labeled a verb, "around" is labeled an adposition (a general term for preopsitions and postpositions), "the" is labeled a determiner, and "lake" is labeled a noun.

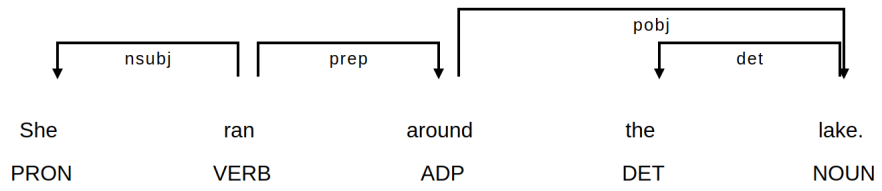


Figure 2.1: Natural Language Syntax Processing

The arcs above the sentence represent the dependency relations between the words. By analyzing this dependency tree we can determine what action is being taken, who is taking it, and the object which the action is being done to. The word “ran” is labeled as the root verb of the sentence, with children “She” and “around”. “She” is labeled as the nominal subject of “ran”, and “around” is labeled as a preposition modifying “ran”. The word “around” has a child “lake”, which is labeled as its prepositional object, and “lake” has a child “the” which is labeled as its determiner.

Another type of syntax parsing which can be performed by NLP is “lemmatization”, replacing the inflected form of a word with its “lemma”, essentially its dictionary form.[8] In this example, the lemma of “ran” is “run”. This would allow us to infer that “runs” in the sentence “She runs around the lake every morning” is the same action as “ran” in “She ran around the lake.”

A particularly interesting and useful development in modern NLP is the creation of word embeddings (or “word vectors”), which represent a word as a large set of numeric values (which comprise a vector in many-dimensional space). This representation is based on that word’s “distributional semantics”, the contexts in which a word is most often used, which can include the other words which surround it or the syntax of the sentence in which it appears. The word vectors that result from this process can be used for a number of different tasks. For example, the distance between two word vectors can be representative of the “similarity” of those two words. Or, words can be added to or subtracted from one another to create a new vector. One common example is that, if the word “king” has the vector for “man” subtracted and the vector for “woman” added, the resulting vector is closest to the vector for “queen”.

2.3.3 NLP tools used in this work

spaCy

To perform dependency parsing and lemmatization, we used spaCy[9]: a Python library which provides implementations of a number of different NLP parsing techniques. In particular, we make use of its implementation of BiLSTM dependency parsing[10] to determine how the words in a sentence relate to one another. BiLSTM, or "Bidirectional Long Short-Term Memory" neural networks are used as a method for generating a representation of a word and the context in which it is used. These are used as a feature function and passed into a non-linear scoring function, which trains the parser along with the BiLSTM itself to infer a good feature representation for the parsing problem (rather than selecting the features by hand). It performs quite well, producing a 93.9 Unlabeled Attachment Score (which represents the percentage of correct dependency parses) on the Penn Treebank Stanford Dependencies dataset, only slightly below the 94.26 record score achieved by Kiperwasser et al.[11] but with a much less complex architecture.

ConceptNet

In order to gain information about how words relate to specific aspects of the physical world, we used ConceptNet[12]. ConceptNet is "an open multilingual graph of general knowledge" which connects words to one another via a set of 34 "relations" such as "Used For", "Created By", or "Made Of". The connections made through these relations allow it to be used to infer how the concepts represented by these words interface with the real world.

ConceptNet makes use of two major sources of knowledge which are not currently being actively updated, Open Mind Common Sense and OpenCyc, along with three which are: DBPedia, Wiktionary, and the Open Multilingual Wordnet. Most of the more specific relations between terms come from the Open Mind Common Sense[13] (OMCS) project. OMCS was a crowd-sourced knowledge base created by presenting contributors with templates like "a hammer is for ___" or "somewhere you find a bed is ___". To ensure the relations collected were of high quality, it also

prompted users to rate pieces of knowledge submitted by others or answer questions to verify existing connections. ConceptNet also makes some use of OpenCyc[14], an effort to manually create a formal representation of the relationships between concepts. The project has since been turned into a private commercial offering, but ConceptNet makes use of the last open release. ConceptNet adds new connections based on DBPedia[15], which extracts knowledge from the infoboxes on Wikipedia articles, and populates most of its synonyms and basic connections between words using Wiktionary and the Open Multilingual Wordnet[16].

Along with direct use of the graph, ConceptNet can also be used to create "word vectors" formatted similarly to those in other embeddings like word2vec[17] which can be directly compared with one another as an estimation of conceptual similarity. In order to create these word embeddings (referred to as ConceptNet Numberbatch), ConceptNet starts by creating vectors using each word's relations as its context. It then performs retrofitting, adding in embeddings from word2vec and GloVe[18], two existing word embeddings created through distributional semantics. When compared on a set of standard word embedding benchmarks to word2vec, GloVe, and another embedding called LexVec, three other state-of-the-art word embeddings, it outperforms (on the MEN-3000, Rare Words, and MTurk-771 benchmarks) or performs equivalently (on the Story Cloze and SAT analogies benchmarks) to all of these other embeddings.

2.4 Related Work

2.4.1 Prior Work

Most prior work in the area of IoT interaction monitoring has been implemented by either processing the code running on a platform [19][20] or by manually creating models of each device's properties, capabilities, and environmental effects [21][22].

Manually creating a model for each device produces a low rate of false positive interactions, but could potentially miss some true positive interactions if the person creating the explicit model fails to consider the effect of some environmental factor. In addition, it requires extra human

modeling effort for every device present in a system and could require manually updating old models if devices with new environmental sensors or effects are added to a platform. For example, if a platform which did not previously support vibration sensors had one added, the "vibration" environmental property would need to be added to devices like washing machines which could cause vibration in the environment.

Automatically creating models by analyzing the raw code interfacing with devices addresses the issue of manual effort in identifying direct and some categories of indirect interaction, but still requires access to the platform code in order to perform analysis. Also, some types of unintentional environmental interaction (for example, a stove heating a room) may not be captured by program analysis which focuses exclusively on devices' intentional environmental interactions.

2.4.2 Concurrent Work

During the development of this work's approach, another work following a similar approach was published, but it uses various strategies to artificially create a set of recipes which they infer would be reasonable for a single user to have enabled, then analyzes the interactions between just those recipes. [23]

Our approach does not involve specifying a set of recipes that a user is "likely to have enabled". As a result, it is able to detect all of its potentially harmful interactions up-front instead of requiring an extra execution every time a user enables a new rule. However, as a side effect of the fact that we do not restrict ourselves to a specific set of recipes, some types of interaction (e.g. "condition bypass", where one recipe performs the same action as another but with fewer conditions) aren't feasible to detect.

In addition, the approach taken in this concurrent work does not check if the environment reaches a set of "undesired states", only if the set of recipes passes a set of internal consistency checks. This means that if a set of interacting recipes is internally consistent, no harm is detected even if the system reaches a state which may be unexpected by or harmful to the user. Our approach, by contrast, allows for the identification of recipe chains which lead to these states.

2.5 Threat Model and Assumptions

2.5.1 Motivation

Trigger-action programs usually have a single trigger and a single action. However, because these programs make changes to the physical environment or other web services connected to the same trigger-action platform, it is possible for the action of one program to cause the trigger of another program to be activated, running another program. This can occur intentionally or unintentionally. For example, one program that causes your alarm system to be armed when you leave home might also trigger another program that turns off the lights when your system is armed.

While this example of chaining is benign and perhaps even intentional, there are other combinations of recipes which might cause more harmful unintended consequences. A recipe that starts a robot vacuum when the mode is set to away may lead to the vacuum activating a presence sensor that activates a recipe that changes the mode back to home, which may activate a recipe that unlocks the front door when the mode is changed to home.

2.5.2 Threat Model

This work intends to address the ways in which a trigger-action platform could be accidentally misused or intentionally abused in a way that goes against a user's intentions or subverts their physical security.

That "accidental misuse" consists of unintentional interactions between otherwise benign recipes, and the "intentional abuse" would most realistically be recipes created with misleading descriptions in order to convince a user to add an automation which would compromise their privacy or security.

Chapter 3

Methodology

The goal of our approach is to extract information from the natural-language descriptions of triggers and actions, then use this information to predict situations in which interactions between trigger-action programs lead to potentially harmful behavior in an IoT system.

3.1 Natural Language Information Extraction

Before we can check for those undesirable properties, we need a way of representing the devices and related trigger-action applications being used in a space in a structured manner. In order to do this, we process the natural language on the titles and descriptions provided by the manufacturer through the following set of steps.

Each of the steps corresponds to a numbered step in Figure 3.1.

Input: the natural language titles and descriptions of each IFTTT Trigger and Action, along with the Trigger-Action pairing associated with each Recipe.

Output: each Trigger-Action pairing annotated with: 1. what in the environment or system can trigger it, and 2. what effects it will have on the environment and system.

As an example, we will take the Recipe “If your smoke alarm detects an emergency, then turn off your oven”. It connects the Trigger “Nest Protect:Smoke Alarm Emergency” to the Action “GE Appliances Cooking:Turn off oven”.

	Channel	Title	Description
Trigger	Nest Protect	Smoke Alarm Emergency	This Trigger fires every time the Nest Protect you specify detects dangerous smoke levels.
Action	GE Appliances Cooking	Turn off oven	This Action will turn off your oven.

Table 3.1: Manufacturer-provided Trigger and Action Information

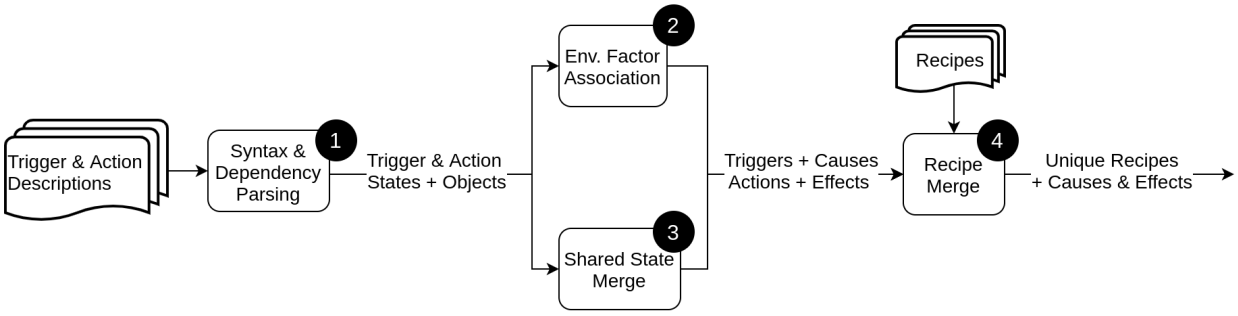


Figure 3.1: Parsing Architecture

3.1.1 Step 1: Syntax and Dependency Parsing

Input: Channel Title, Trigger or Action Title, Trigger or Action Description

Output: the mapping of each Trigger and Action to the relevant Object and State

Procedure:

For the Title and Description assigned to each Trigger and Action, we use spaCy[9] to extract the dependency relations between the words in each sentence and their parts of speech. Figure 3.2 displays the dependency relations between the words in our sample Trigger, and Figure 3.3 displays the relations between the words in our sample Action.

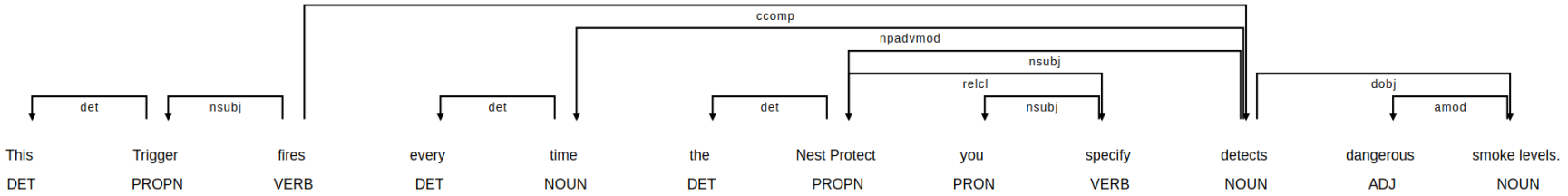


Figure 3.2: Trigger Dependency Graph

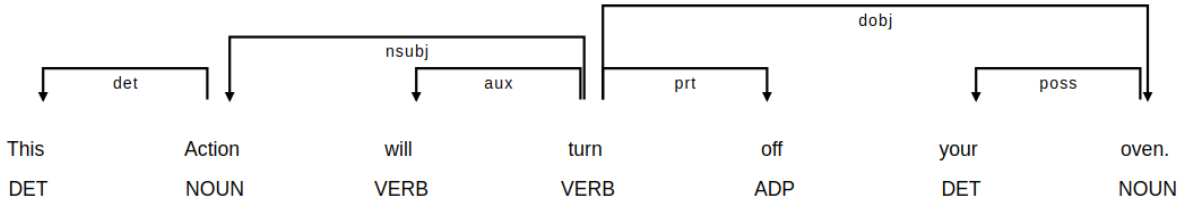


Figure 3.3: Action Dependency Graph

Starting with the root verb in the sentence (“fires” in the case of the trigger and “turn” in the case of the action), we follow the dependency arcs to the other words in the sentence and check each sequence against a list known to correspond to objects and a list known to correspond to actions.

After locating the state and object, we add the lemmatized version of each matching word to the trigger/action’s list of states or objects. Using the lemmatized version allows us to more consistently associate these across triggers and actions later in the process.

Analysis

Because of the significantly different sentence structures, the arc sequences which led to the relevant object and state words in triggers were entirely different from those used in actions. Trigger states were usually clausal complements (ccomp) or adverbs in various relations with the root verb, and their associated objects were either the direct subject (dsubj) of the root verb or the subject/object (nsubj/dobj) of the trigger’s state. Action states were usually the root verb itself, and their associated objects were either that state’s direct object (dobj) or its conjunct (conj). In the case both Triggers and Actions, if the State word or the Object word had a participle (prt, like “off” in the case of “turn off”), we also included that in the list of state words in order to infer a direction for the action.

In the case of the Nest Protect smoke alarm, the “ccomp” dependency arc from the root verb to the word “detects” matches a pattern for the Trigger State, and the “dobj” arc from the State word “detects” to the compound noun “smoke levels” matches a pattern for the Trigger Object, as does the “nsubj” arc from “detects” to “Nest Protect”. In the case of the GE Oven, the sentence structure is significantly simpler. The root verb “turn” is the Action State, and the word “off”, connected via the “prt” relation to the root verb, is also included in the list of State words. The word “oven” is connected to the State by the “dobj” arc, which matches a pattern for the Action Object.

So, at the end of the Syntax and Dependency Parsing step, we have extracted the following data:

	State	Object
Trigger	detect	nest protect, smoke levels
Action	turn, off	oven

Table 3.2: Syntax and Dependency Parsing Output

3.1.2 Step 2: Environmental Factor Association

Input: the Object and State associated with each Trigger or Action

Output: For each trigger, all of the environmental factors which might cause that trigger to fire, and the direction (increasing, decreasing, or toggled) in which the factor would need to change for the trigger to fire. For each action, all of the environmental factors which that action might affect and the direction of that effect.

Procedure:

The environmental factor association process can involve between one and four steps of comparison between words describing the Object and words describing each environmental factor. This process is shown in Figure 3.4.

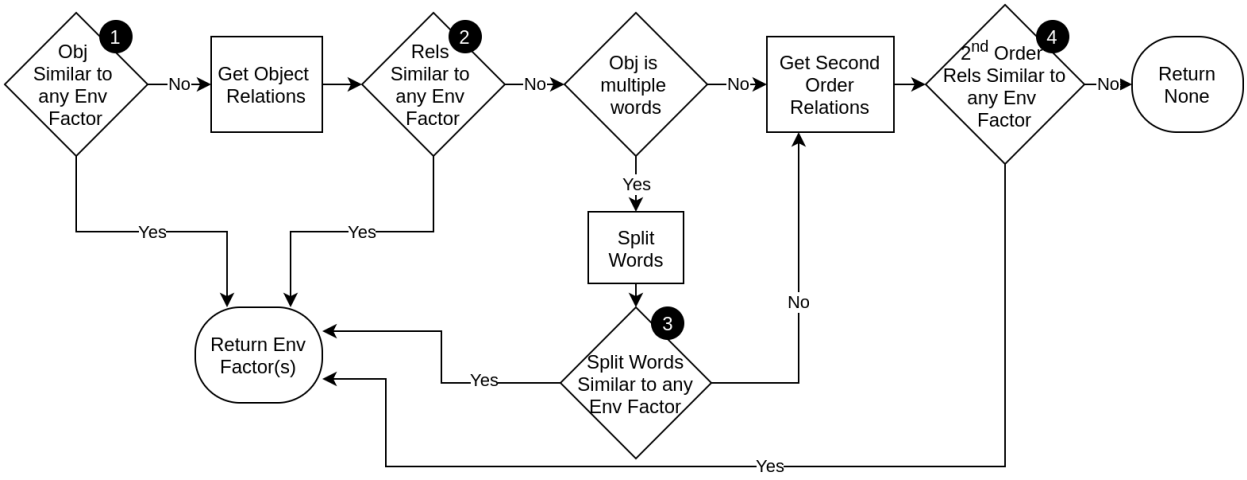


Figure 3.4: Environmental Factor Association Process

Comparison 1: For each object inside each Trigger and Action, we start by directly comparing the word vector representing the object to the word vectors of each environmental factor.

If none of these environmental factors has a similarity greater than 0.42 (a manually determined threshold based on the frequency of false positive associations), we progressively expand the search. The next step is to follow that object's relations in the ConceptNet database.

Comparison 2: We compare each environmental factor to each of the words connected to our object by the "Capable Of", "Used For", "Has Property", and "Causes" relations.

If comparing these first-order relations to the environmental factors yields no results with a sufficiently high similarity and the object had fewer than five total relations across those four categories, the object is not well-connected to the rest of the words in the net. This often occurs when an "object" we have extracted consists of multiple words.

Comparison 3: So, if we have multiple words, we compare the similarity of each of these individual words with the environmental factors.

Comparison 4: If all of those steps fail to find a sufficiently related environmental factor, we make one last attempt by checking the similarity of the object's second-order relations.

If none of the second-order relations have a similarity to an environmental factor which passes the threshold, we do not return any environmental factors. We can be fairly sure that the object is not closely related enough to an environmental factor to have an effect on it.

In conjunction with the association of each object to environmental factors, we compare the "state" word against lists of words which are known to signify an "increase", a "decrease", or a "toggle" in the state. This tells us which direction the object is sensing or acting in and allows us to more accurately infer when a real interaction will occur between an action and a trigger.

Analysis

Based on the things in the environment which the IoT devices in this dataset were capable of sensing, the environmental factors that we analyzed for potential interactions were light, temperature, humidity, water, sound, motion, air quality (which includes CO and CO2), and smoke. Using ConceptNet's vector embeddings[12], we collected the set of the 200 words most associated with each of these environmental factors, with the limitation that the vector similarity had to be above

0.5 (another manually determined threshold based on the frequency of false positive associations). We used fewer for humidity, air quality, and smoke, because the vector space near them was sparse enough that 200 words encompassed some unrelated objects and concepts.

While ConceptNet contains 34 different types of relations, the ones which most reliably expressed how an object interacts with its environment were “Capable Of”, “Used For”, “Has Property”, and “Causes”. The word “oven”, for example, is connected via “Capable Of” to “bake”, “roast”, and “heat a meal”, by “Used For” to “cooking”, “baking”, and “roasting”, and by “Has Property” to “very hot” and “powered by burning gas”.

Maintaining a relatively high similarity threshold and following a small number of highly relevant relations allowed us to expand our detection of potential interactions without introducing nearly as many false positives as would result from simply lowering the similarity threshold.

In the case of our example, the Object “smoke levels” in our Trigger had a similarity of 0.715 to the environmental factor “smoke”, and the Object “oven” in our Object had a similarity of 0.566 to the environmental factor “temperature”. The Object “Nest Protect” in our Trigger, however, did not have a similarity to any of our environmental factors which passed the threshold for inclusion, and so did not change the Trigger’s environmental associations.

When the State words are checked against the list of words signifying an “increase” or “decrease”, we find that the “detect” State word in our Trigger is associated with an “increase” in a factor, and the “off” State word in our Action is associated with a “decrease”.

Now, at the end of the Environmental Factor Association step, we have extracted the following data:

	State	**Object**	**Env.**	**Direction**
Trigger	detect	nest protect, smoke levels	smoke	increase
Action	turn, off	oven	temperature	decrease

Table 3.3: Environmental Factor Association Output

3.1.3 Step 3: Shared State Merge

Next, we identify whether this trigger or action have any relationship to a set of "shared states", and, if so, mark the trigger or action with which one of those states it affects.

Input: the Object and State associated with each Trigger or Action

Output: the change in a set of "shared states" which would cause each Trigger to fire or the effect each Action will have on that set of "shared states"

Procedure:

In addition to determining what environmental factors a trigger or action might be associated with, we check whether each trigger or action's state is associated with a "lock"/"unlock" event, a "home"/"away" event, an "arm"/"disarm" event, or an "open"/"close" event. Each of these, while not strictly a feature of the environment in the way that temperature or water are, changes a state which is relevant to the operation or security of the rest of the system. Therefore, if any of these are present in the "state" list, we move them to a separate "shared states" list.

Analysis

In the case of our example, neither the Trigger nor the Action had any relation to the Shared States, so this step does not contribute any information.

3.1.4 Step 4: Recipe Merge

For the final step in our parsing, we take all of the information we have extracted regarding the Triggers and Actions and attach them to the user-created Recipes which combine them, creating a set of unique trigger-action combinations which we will check for potential harmful interactions.

Input: the set of 300,000 IFTTT Recipes

Output: every unique Trigger-Action pair with all of the extracted information attached

Procedure:

Because we have extracted information about the effects of each Trigger and Action, we just need to consider the unique Trigger-Action pairs among the dataset of Recipes. For each Recipe, we extract the Trigger title, Action title, Recipe name, and the number of users who have enabled it. If the trigger-action combination is not yet present in our list of unique pairs, we create an entry and populate it with the processed trigger and action along with the recipe's add count, then add the recipe title to that trigger-action pair's entry. If the pair is already present, we just add the recipe's title to that trigger-action pair's entry and add the recipe's add count to that of the trigger-action pair.

Analysis

In our example, this creates an entry for the Trigger-Action tuple ('Nest Protect:Smoke alarm emergency', 'GE Appliances Cooking:Turn off oven'), which fires on an "increase" in the "smoke" environmental factor and causes a "decrease" in the "temperature" environmental factor, and it then associates the recipe 'GE Appliances Cooking:If your smoke alarm detects an emergency, then turn off your oven' with that tuple, incrementing its "add count" by 8.

3.2 State Graph Generation

3.2.1 Undesirable Properties

In an IoT environment, a number of different circumstances can arise in which the system behaves in an unexpected or dangerous way. Some of these undesirable properties which have been explored in prior work, such as improperly written code [19] or those with user-defined input (such as activating at a certain time) are not feasible to detect without access to a manufacturer's code or an individual user's private settings. However, the presence of a number of undesirable properties can be ascertained through analyzing the publicly available natural language descriptions of

triggers and actions. The properties which we focus on in this work are:

- reaching an undesirable state (for example, if a user is away but their door is unlocked)
- contradicting the initial recipe's intentions (the initial recipe turns on a light, but a recipe later in the chain turns that light off)
- double-firing action (an action is executed twice in the same chain: e.g. door is unlocked twice, watering is started twice, etc.)

3.2.2 Graph Generation

The overall goal of this state graph generation is as follows: from a set of known initial states, determine the possible chains of recipes which follow from a change in a single environmental factor or shared state and ascertain whether these chains lead to unintended and undesirable properties in the IoT system.

While other approaches intended to check whether safety properties are violated use formal model checking in order to see whether there is any path which violates one of those properties, this is not suited to the goal of our approach. Instead of checking a single property or a single set of recipes, the goal of our approach is to find all instances of multiple types of potential safety property violations across an entire dataset. Because there is no single heuristic which could direct us to all of the different types of violations, our approach performs a breadth-first search of the entire state set. This limits the depth to which it can search, but this trade-off enables graph generation to be performed once and then applied to all possible users of the platform regardless of which recipes (or which combination of recipes) they have enabled.

Input: Unique trigger-action pairs annotated with shared states and environmental factor associations

Output: Sequences of trigger-action pairs implicitly connected via environmental factors or shared states which break one of the following safety properties:

Procedure:

Start in a "known safe" state with the user home, their alarm system armed, their door locked and shut, and all environmental states in a "neutral" position. Then, change each shared state and environmental factor in turn, starting a separate chain for each recipe which fires as a result of that change in environmental or shared state. For each new chain, record the environmental and shared-state effects of the first recipe's action (both the factor and the direction in which it is being changed) Then, repeat the following until no further recipes are triggered or a certain chain length is reached:

1. record any shared states induced by the action which are different from the current states
2. for every environmental factor change induced by the action
    - if the action explicitly increases or decreases a factor and the factor is not already in that state, change it to high (if it increases) or low (if it decreases) and record that change
    - if the action toggles a state and the state is currently high or low, flip to the opposite state and record that change
    - if the action has an unknown effect on the factor or we are toggling a factor which is currently neutral, split into two new chains: one in which the factor has been increased and the other in which it has been decreased.
3. for each of the possible new states (there will be only one if we are sure what its effects on the environmental factors will be), check these new states against the safety properties. If it violates any of them, store the sequence of recipes and resulting state changes which led to that violation.

Analysis

Continuing with our example from the Information Extraction step, one of the sequences of potential interactions the State Graph Generation step infers is displayed in Figure 3.5. In this

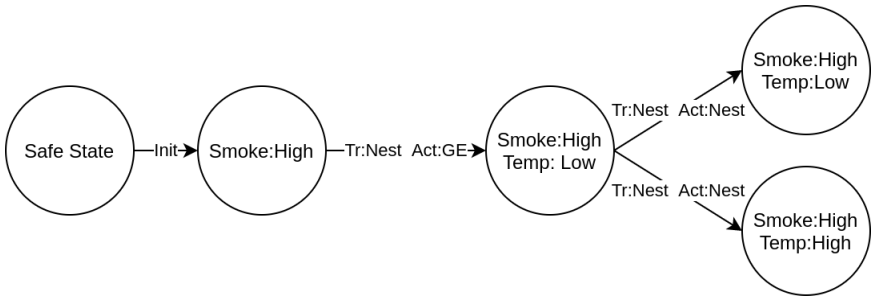


Figure 3.5: Example Subgraph

case, we start in the known safe state, then change the environmental factor "Smoke" to "High".

This increase in Smoke will cause the "Nest Protect:Smoke alarm emergency" trigger to be activated, and fire the "If your smoke alarm detects an emergency, then turn off your oven" recipe, which will cause the "GE Appliances Cooking:Turn off oven" action to be activated. Because "oven" was related to a decrease in the Temperature environmental factor and not Smoke, this recipe moves the graph to a new state where the level of Smoke is still High but the Temperature is now Low.

This decrease in temperature will then activate triggers which respond to a decrease in temperature. One of these triggers is "Nest Thermostat:Temperature drops below", which is connected by a recipe to the action "Nest Thermostat:Set temperature". This action sets the temperature to a user-specified level, and was classified by the Environmental Factor Association component of the parser as having an "unknown effect" on the Temperature environmental factor.

As a result, the chain splits off into two new chains: one in which the "Nest Thermostat:Set temperature" Action caused the temperature to increase, and one in which it caused the temperature to decrease.

This chain would then continue until it either violated one of the safety properties or reached the maximum search depth.

Chapter 4

Evaluation

In this evaluation, we studied the degree to which we can extract meaningful semantics to determine whether there are harmful interactions. In this, we asked three fundamental questions:

1. Can implicit interactions between IoT devices be identified realistically?
2. Do the resulting identified interactions represent actual harm?
3. Is the process fast enough that it can be applied in the real world?

4.1 Dataset

To evaluate the performance of our approach on real-world trigger-action programs, we used the IFTTT dataset collected by Mi et. al.[24] in order to analyze the ways in which trigger-action integrations between different service and device providers were being utilized by real users. This dataset consists of 300,000 user-created recipes, comprised of 17,000 unique pairs of 1,470 triggers and 896 actions. Each trigger and action contains the title and description created by the device manufacturer or web service operator, which we assume are truthful. Each recipe contains a reference to one trigger and one action, along with a count of how many users have enabled the recipe and a user-created title and description. We record the recipe title and description, but do not

take them into consideration for inferring the recipe's effects since we do not consider unknown users of the IFTTT platform to be trusted parties.

4.2 Identifying Implicit Interactions

4.2.1 Parsing

We found that our approach was able to successfully extract object and state transition information from almost all of the trigger and action descriptions included in the dataset. The following is an example of how our parser enables this.

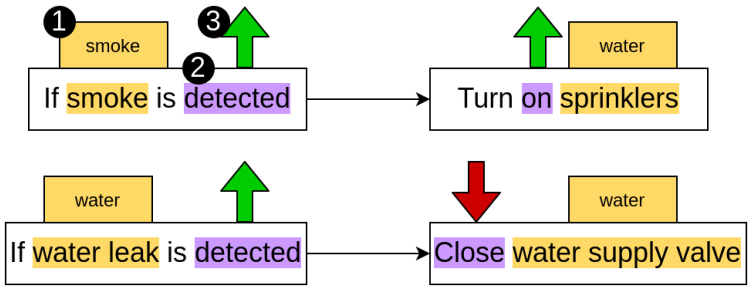


Figure 4.1: Water Leak Example

In this example, the parser in our approach:

1. extracts the word corresponding to the trigger or action's object and associates it with environmental factors
2. extracts the change in state which the trigger fires on
3. infers the direction of state which a trigger fires on

This is performed for each trigger and action in the dataset. Once the parsing is completed, the state graph checker determines which recipes will fire as a result of a change in any environmental factor. In this case, the first recipe would fire as a result of an increase in "smoke". The action of the first recipe ("Turn on sprinklers" in this case) is stored as the "intention" of the recipe, which should not be undone by other recipes that might be chained with this one. The effect of this first recipe is then

applied to the environment (meaning that both the “smoke” and “water” environmental factors are in a “high” state). The graph checker finds each recipe that will be fired as a result of this change in the detected level of “water”, which includes the second recipe shown here, which triggers on a water leak detector. The action of this second recipe is to close the water supply valve in order to stop the water leak and prevent flooding. However, this is not the action which a user would like the system to take because it would cause the sprinklers to stop functioning. This is reflected in our approach’s model in two ways. First, as a “harmful state” in our model: “smoke” is in the “high” state and “water” is “low” state. Second, it is detected as a recipe contradiction: the effect of the first recipe was to increase the amount of water, and the effect of the second is to decrease the amount of water.

4.2.2 Detection of Harmful Paths

In order to enable the detection of potentially harmful interactions between IoT devices, we create a graph representing how a state of the IoT system could be reached through environmental or shared-state interactions. We check each state along with the chain of states necessary to reach it for three properties: harmful individual states, duplicate actions in a chain, and contradictions of the initial recipe.

In the table of results, the following terms are used:

Nodes: the number of specific environmental states (aggregated and deduplicated across all searches)

Edges: the number of links to a new environmental state which could be reached by triggering a recipe (this is the sum of new states checked across all searches, with no deduplication across different chains)

Harmful States: the number of chains by which we can reach a state explicitly described as “harmful”. For the purposes of this evaluation, the harmful states used were “user is away, door is unlocked”, “user is away, security system is disarmed”, “smoke is detected, water is turned off”, “smoke is detected, alarm is silenced(sound is low)”, “user is away, there is motion, and there is

no audible alarm", "user is away, lights are on", and "user is away, door is open".

Duplicate Actions: the number of chains where a different recipe with the same action was fired as a result of a previous recipe in the chain.

Recipe Contradictions: the number of chains where, after the first recipe was triggered, a subsequent recipe reverted the change made by that first action.

	Depth 1	**Depth 2**	**Depth 3**	**Depth 4**
Time (sec)	1	30	767	15301
Nodes / Edges	87 / 71091	409 / 2002610	1259 / 46062367	2785 / 823021991
Harmful States	1	21	559	19548
Duplicate Actions	0	258	11896	323062
Rec. Contradictions	0	34	540	15560

Table 4.1: Results of Harmful Paths Search

4.3 Analysis of Identified Interactions

Our analysis of the repercussions of interactions between these IoT devices led to some concerning potential scenarios in each of the path categories we flagged.

4.3.1 Harmful States

Our comparison of each visited state against a set of states with known harmful properties led to the detection of a number of concerning interactions.

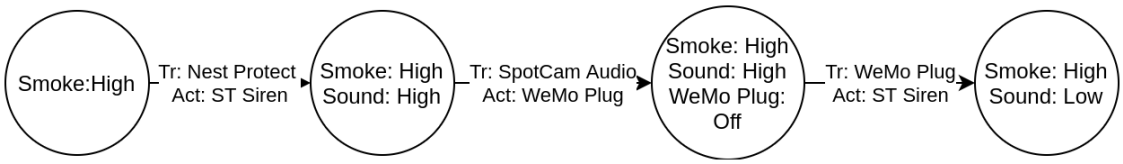


Figure 4.2: Violation of "Smoke: High, Sound: Low" safety property

In one instance depicted in Figure 4.2, a Nest Protect smoke alarm emergency was intended to activate a SmartThings siren/strobe. However, this siren activated another recipe which switched off a WeMo Smart Plug when a SpotCamHD detected an audio event, and a recipe following that

deactivated the SmartThings siren/strobe as a result of the WeMo Smart Plug being switched off. This led to the violation of the Recipe Contradictions property as well as the Harmful State rule against an alarm being switched off when there is smoke.

In another case, a Ring doorbell being pushed triggered a Philips Hue lightbulb to blink. This activated a recipe which caused a SmartThings plug to turn on when SmartThings detected an increase in brightness, which then violated the "away and lights on" Harmful State rule. The degree of danger this poses depends on what the SmartThings plug is attached to, but this situation leads to anyone outside a user's house being able to turn on a device by ringing their doorbell.

4.3.2 Duplicate Actions

"Duplicate Actions" represented, in quantity, the largest number of detected harmful paths. Most of these paths consisted of lights and furnaces/AC units performing repeated toggles through the intermediates of temperature, light, or sound. Particularly with temperature, the lack of ability to determine an actual threshold value for the triggers or actions led to a number of instances where the environmental model's temperature would oscillate between cold and hot, triggering recipes in response to the change each time.

4.3.3 Recipe Contradictions

The "Recipe Contradictions" rule was able to detect a large number of potentially harmful interactions, including detection of some of the same interactions as the "Harmful States" without needing to have those specific harmful situations enumerated.

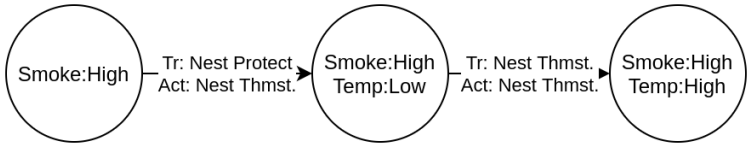


Figure 4.3: Self-Contradictory Furnace Switching

One interesting contradiction detected, shown in Figure 4.3, was the combination of a recipe which decreases the temperature when a smoke alarm goes off with a recipe which increases the

temperature once the temperature drops below a threshold. If a user's smoke alarm is going off, they may want the first recipe to turn off their furnace so it does not increase airflow to the fire. However, if the second recipe is also enabled, it will be triggered and will re-start the furnace, negating the potential protection provided by the first recipe.

In a similar but less harmful vein, there were a number of other potential interactions where, if "vacation mode" is triggered for some device in a system but other temperature-maintenance recipes are enabled, these will bypass the vacation mode and change the temperature anyway.

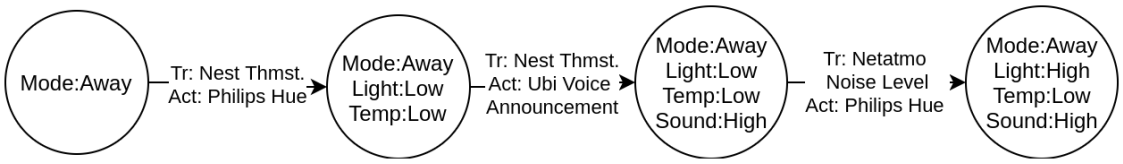
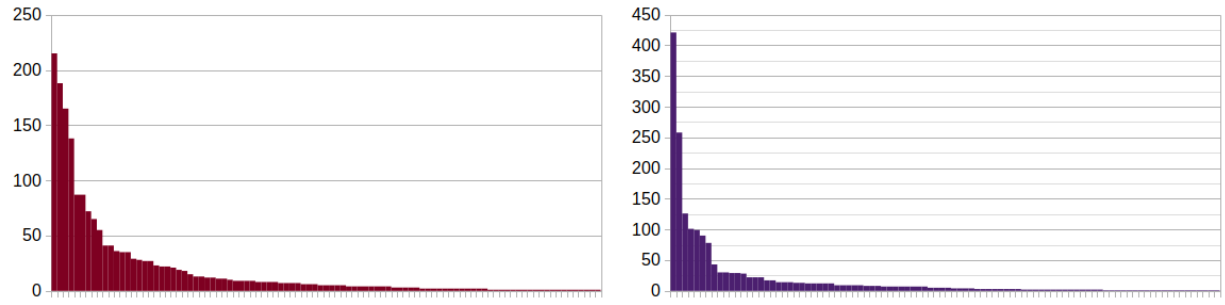


Figure 4.4: Self-Contradictory Light Switching

There were also a number of instances where a smart thermostat being set to "away" mode would automatically turn the lights off, but then if that caused the temperature to drop then another recipe would fire which would turn the lights back on. In one case, as seen in Figure 4.4 the low temperature would directly turn a light back on, but there were others which, for example, gave a voice announcement when the temperature was low, triggering a noise sensor, which then caused the lights to turn back on.

4.3.4 Analysis of Interaction Categories

We found that the majority of implicit interactions detected in this dataset by our approach were through the light and temperature environmental variables. This makes some sense, because these are often useful things to trigger recipes on, but the fact that they made up such a large proportion of the detected interactions is possibly a result of the available recipes in the 2017 dataset used for this analysis. Table 4.2 is a list of the IoT Trigger and Action channels with the highest number of unique recipes. All of the entries other than SmartThings are explicitly related to lighting and heating, and many of the recipes making use of the SmartThings channel affect or are affected by



(a) Recipes per Trigger Channel

(b) Recipes per Action Channel

Figure 4.5: Distribution of Trigger and Action Recipes by Channel

light or temperature.

Trigger Channels	Action Channels
Nest Thermostat: 215	Philips Hue: 421
SmartThings: 188	WeMo Smart Plug: 258
Netatmo Weather Station: 165	SmartThings: 126
Nest Protect: 138	WeMo Insight Switch: 101
WeMo Smart Plug: 87	Nest Thermostat: 99
WeMo Light Switch: 87	LIFX: 90

Table 4.2: Number of Unique Recipes by Channel (IoT-only)

In addition, after these most common channels, the usage of less common channels drops off dramatically, with many channels only having unique recipe counts in the low single digits. This distribution can be seen in Figure 4.5. Because interactions were only considered if they were enabled by a recipe in the dataset, many devices which likely would interact if used together in a physical IoT environment were not able to be captured by this analysis.

4.4 Performance

4.4.1 Graph Search Performance

As would be expected from a breadth-first search, the execution time, number of state nodes, and number of recipe edges searched grows roughly exponentially with the increase in search depth. The search could be easily parallelized, since each chain of recipes branches off into an

independent search, but since that would only decrease the execution time by a fixed multiple these growth rates are still representative of what could be expected in a real-world execution. While these growth rates are quite significant, unlike other approaches this search will only have to be run once for any set of triggers, actions, and unique trigger-action combinations. This growth rate will also limit the depth of a feasible search. However, as a result of the fact that there is a possibility of a false-positive interaction (either one that does not exist in real life or one that does not apply to a user), every increase in depth comes with a significant drop-off in utility to real users.

Estimated execution time growth (sec): $0.04428e^{3.21483x}$**,** $(R^2 = .99919)$

Based on the line's fit to the data points, this exponential appears to accurately estimate time growth.

Estimated state node growth: $33.34021e^{1.15226x}$**,** $(R^2 = .97898)$

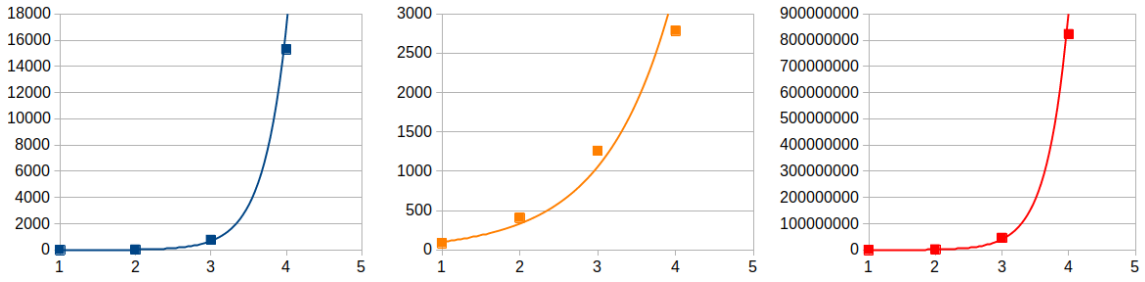
Based on the line's fit to the data points, this exponential appears to overestimate node growth. It makes sense that the node growth rate would decrease before the edge growth rate, since there are a significantly smaller number of states than there are recipes which allow for movement between them.

Estimated recipe edge growth: $3506.76847e^{3.12059x}$**,** $(R^2 = .99893)$

Based on the line's fit to the data points, this exponential appears to slightly overestimate edge growth. Although there are also a finite number of edges which could connect the state nodes, the number of possible edges is large enough that we do not reach saturation by depth 4.

4.4.2 Association Performance

The initial parsing and association of the 300,000 recipes, 1,470 triggers, and 896 actions takes 120 seconds. The parsing and analysis both execute independently of the graph search and can be



(a) Execution time (b) Num. of nodes (c) Num. of edges

Figure 4.6: Graphed time, node, and edge counts for depth 1-4

easily cached for future use, and their execution times are insignificant in comparison to that of this approach's other elements. This will continue to be the case as the popularity of trigger-action platforms grows, since execution time grows sub-linearly with the size of the dataset. A number of recipes, triggers, and actions share words describing objects and states which can be cached to avoid multiple factor-association lookups. These lookups are the most computationally expensive piece of the association, as they include both the ConceptNet relation lookup and multiple calculations of vector similarity between words.

Chapter 5

Discussion and Limitations

5.1 Application of Results

In order to apply the collected graph to individual users of the platform, the pre-processed graph can be intersected with a user's set of enabled recipes. For the set of recipes that a user has enabled, the paths can be followed in parallel, then after each step the resulting state of the nodes each of their recipes reached can be merged. If any of the paths remaining causes a violation of one of the safety properties, the user can be made aware of this interaction and can make an informed decision as to whether they want to enable that set of recipes.

5.2 Comparison

Unlike other tools which analyze the interaction between IoT applications[19], our approach does not rely on app code in order to generate its model. Although this can reduce the accuracy of its predictions, our approach is applicable to the model of third-party integration of devices from multiple manufacturers that is dominant in today's IoT landscape. While other tools require constant re-execution inside each user's environment to determine whether a property might be violated in the near future [21], our approach performs a single pass across an entire dataset. There

are other approaches which do not rely on constantly checking the current state of the IoT system, but do require re-execution of the entire modeling process for every user any time they change the set of recipes they have enabled[23]. However, our approach has to perform this modeling only once, which could make it more suitable for integration providers like IFTTT which serve millions of users with frequently changing and unique recipe sets.

5.2.1 Graph Search Approach

While our approach's undirected breadth-first search is slower than other approaches and difficult to scale beyond a depth of 4, it is still suitable for detecting and pre-processing all possible harmful interactions within that depth in a number of categories. An approach often taken in projects seeking to determine whether particular states can come about in an environment is formal model checking, where a state and a set of transition functions is presented to an Satisfiability Modulo Theory (SMT) solver to determine whether a particular state can be reached. While this is appropriate for situations in which one wants to discover whether it is possible for a single type of condition to be met within a system, it is less applicable when it is assumed that there are many ways to meet those conditions and the goal is to find the complete set of paths by which they can be met. In addition, while more informed search algorithms could be created for finding a single type of vulnerability (for example, reaching a particular undesirable state) based on a heuristic, that would likely cause the search to miss occurrences of other property violations.

5.3 Limitations on Dependency Parsing

Our dependency parsing is separated from our attempts at gaining a conceptual understanding of the words. As such, any incorrect parsing that stems from a lack of understanding of the world would be extremely difficult to solve.

In some cases, the grammar parsing runs into issues where a sentence could reasonably be parsed in multiple different ways, and the correct parsing depends on extra knowledge about the

meaning of words in particular contexts: see Figure 5.1

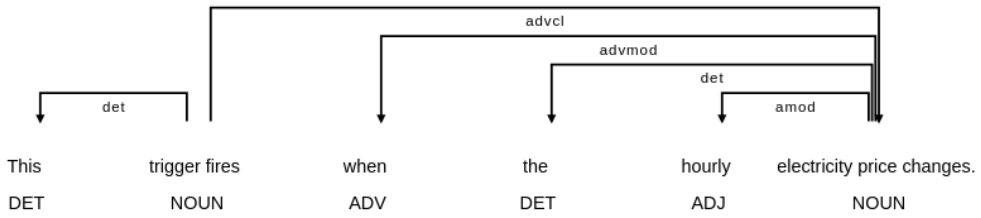


Figure 5.1: Noun-Verb Ambiguity

In this case, “electricity price changes” could be interpreted as a single noun, meaning “changes in the price of electricity”, or as a verb phrase: the “electricity price” (noun) “changes” (verb).

5.4 Limitations on Conceptual Association

The ConceptNet relations were very helpful in associating devices with environmental factors. However, its data is taken largely from the Open Mind Common Sense database, which consists of a set of human-generated responses to questions about the relationships between objects. The fact that the relations were created by a large number of contributors lead to a number of quirks in the relations between objects and environmental factors. For example, “glass” was interpreted by some ConceptNet contributors as “drinking glass” and by others as “window glass”, and since a “door” often has a glass window, that relation was combined into the same word entry, meaning that it wasn’t possible to automatically disambiguate.

5.4.1 Dataset quality

While the dataset used in the evaluation of our approach was large and represented a wide variety of IoT device types, there were some issues which led to difficulties in determining recipe chains. For example, there are entries in the set of triggers which refer to August smart locks but no corresponding entries in the set of actions, and while LinkedIn is present in triggers/actions

inside recipes, it is not in the set of triggers. While these occurrences were rare, the fact that there were multiple issues of this type is worth noting.

5.5 Generalizability

5.5.1 Cross-Platform Generalizability

All trigger-action platforms suffer from the same lack of visibility across manufacturers and devices, meaning that the type of NLP-driven analysis used by our approach remains the most informative source of data regarding the emergent behavior of the IoT environments it enables. The approach used by this work, along with almost all of its code, are directly applicable to trigger-action platforms more generally.

5.5.2 Cross-Language Generalizability

ConceptNet and spaCy are capable of providing relations and processing text across multiple languages, but the grammatical differences between different languages would likely require manual adjustment of the dependencies spaCy uses to extract the relevant pieces of sentences on a per-language basis.

5.6 Impact of Physical Environment Configuration

While the physical location of devices in relation to one another will have an impact on how and to what extent they interact with each other, it is not feasible to infer the locations of devices involved in the recipes based on the information available in the trigger and action descriptions. This could be attempted using recipe descriptions in some cases, but a recipe is often created by one user and enabled by hundreds or thousands of other users which might have their devices in different locations. If an approach like ours, which detects potential interactions through environmental modeling, were implemented by a service provider, they could potentially request that a

user provide the relative locations of their devices. However, this would likely introduce significant complications in pre-processing all of the associations.

5.6.1 Outdoor vs. Indoor Events

One particularly interesting example of physical environmental interaction was weather forecast triggers. If the temperature outside is high, that will likely have an effect on the internal temperature of the house. However, that increase in temperature is not due to anything in the IoT system. This model has no way of distinguishing between those things, however, and so it led to a number of false positives. This issue could likely be alleviated by marking some trigger-action channels as "external input" and prohibiting them from being an intermediate link in the chain during analysis.

Chapter 6

Conclusions

6.1 Results

In this thesis, we explored extracting information from natural language about the ways in which triggers and actions assigned to IoT devices interact via their environment in unwanted and potentially dangerous ways. In particular, we investigated an approach which attempts to use the natural language descriptions already present on trigger-action platforms to pre-determine all possible instances of a number of harmful interactions. We evaluated this approach's performance against a large set of real-world IoT trigger-action recipes, discovering a large number of potentially harmful recipe chains, and demonstrated the feasibility of performing checks for multiple types of harmful interactions concurrently on multi-recipe chains.

However, we also discovered that this approach is limited by performance in the chain length it is capable of searching, and we discovered that limiting the triggers and actions under consideration to those which are already in use can lead to challenges in extracting interaction information from a dataset.

6.2 Future Work

This NLP-based approach provides a foundation for further work in enabling the analyses which will be necessary to ensure the safety and reliability of IoT environments as the field expands in the future. To further advance the usefulness and applicability of this approach, it would be important to consider improvements both to the parsing and to the graph search.

It would be worthwhile to investigate automating the selection of the dependency trees used for identifying objects and states, along with the relations and thresholds used for environmental factor association, perhaps through an approach based on machine learning. This would potentially lead to even more accurate parsing, and would decrease the manual effort required modify a tool to work with descriptions written in a language other than English.

It may also be helpful to investigate whether some sort of informed graph search or state pruning could be applied to the problem of interaction identification in place of the search used in this work while preserving the rate of true positive interaction identification.

6.3 Necessity of Automated Harmful Interaction Detection

As IoT becomes more widely adopted and the devices we all have inside our homes become more varied, these sorts of interactions are only going to become more commonplace. Unfortunately, collections of discrete trigger-action programs do not on their own provide any means of detecting to or responding to interactions between those programs. A platform on which manufacturers provide accurate descriptions of the types of environmental factors their devices interact with may make parsing natural language less necessary (at the expense of increased human effort for the creation and maintenance of these lists). Alternatively, a closed platform might be created where the manufacturer is able to determine before production the ways in which their devices might interact and program checks to detect these interactions into the software of the system itself. However, although there may be variations in the implementation, the task of automating the

detection of potential interactions will likely remain relevant for many years to come.

Bibliography

[1] Philips. Philips hue lighting. https://www2.meethue.com/. 6

[2] GE. Ge connected appliances. https://www.geappliances.com/ge/connected-appliances/. 6

[3] Arlo. Arlo smart home security cameras. https://www.arlo.com/. 6

[4] IFTTT. About - ifttt. https://ifttt.com/about. 6, 7

[5] Samsung. Smartthings apps. https://www.samsung.com/global/galaxy/apps/smartthings/. 6

[6] Apple. Create home automations with the home app. https://support.apple.com/en-us/HT208940. 6

[7] Samsung. Smartthings classic app transition. https://support.smartthings.com/hc/en-us/articles/360034460812. 6

[8] Edward Loper Bird, Steven and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009. 10

[9] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017. 11, 16

[10] Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016. 11

[11] David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. Structured training for neural network transition-based parsing. 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)*, pages 323–333, Beijing, China, July 2015. Association for Computational Linguistics. 11

[12] Robyn Speer, Joshua Chin, and Catherine Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge, 2016. 11, 20

[13] Push Singh. The public acquisition of commonsense knowledge. In *AAAI Technical Report SS-02-09*, 2002. 11

[14] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989. 12

[15] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 12

[16] Francis Bond and R. Foster. Linking and extending an open multilingual wordnet. *51st Annual Meeting of the Association For Computational Linguistics: ACL-2013*, pages 1352–1362, 01 2013. 12

[17] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. 12

[18] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. 12

[19] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference*, pages 147–158, 2018. 12, 23, 36

[20] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 832–846. ACM, 2018. 12

[21] K. Hsu, Y. Chiang, and H. Hsiao. Safechain: Securing trigger-action programming from attack chains. *IEEE Transactions on Information Forensics and Security*, 14(10):2607–2622, Oct 2019. 12, 36

[22] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1501–1510. International World Wide Web Conferences Steering Committee, 2017. 12

[23] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1439–1453, New York, NY, USA, 2019. Association for Computing Machinery. 13, 37

[24] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of ifttt: Ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 398–404, New York, NY, USA, 2017. Association for Computing Machinery. 27

Academic Vita

Jonathan Bees

Education
B.S. Computer Science at Penn State University - Schreyer Honors College, Grad. May 2020

Skills
Cybersecurity: Offensive Security Certified Professional, experience with IDA and Carbon Black
Software Development: Java, C, Python, Groovy, SQL
Containerization/Orchestration: Docker, Kubernetes
Networking: Wireshark, OpenVPN, 802.1x/RADIUS
VM Configuration/Management: KVM/VirtualBox/VMWare

Work Experience
IBM – Security Services Specialist Intern (2019)
- Initiated transition of an internal malware analysis platform to a scalable microservice architecture
- Reverse-engineered malware using IDA to develop parsers for automated sample deobfuscation
- Developed Carbon Black queries incorporated into detection of malicious activity in IBM customers' networks

Oracle - QA Engineer Co-Op (2018)
- Developed automated testing suites in JUnit and Selenium, maintained Jenkins CI pipeline
- Started project to migrate single-host software platform to Docker/Kubernetes

Penn State Lunar Lion Team - Command & Data Handling Lead (2016-2018)
- Designed system on Raspberry Pi hardware platform for telemetry and craft control
- Developed Command & Data Handling software built on NASA's Core Flight Executive

Carnegie Mellon University CyLab Usable Privacy and Security Lab – Intern (2014)
- Performed research on Human Factors/UX in password security
- Member of CMU's 2014 "Crack Me If You Can" password-cracking team – division champs

Publications
Blase Ur, **Jonathan Bees**, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor. 2016. *Do Users' Perceptions of Password Security Match Reality?* In proc. Association for Computing Machinery

Blase Ur, Fumiko Noma, **Jonathan Bees**, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor. 2015. *"I Added '!' at the End to Make It Secure": Observing Password Creation in the Lab.* In proc. Usenix

Extracurriculars
Bee House THON organization (2016-present)
- Fundraising for pediatric cancer research & family support

FIRST Robotics Competition team member (2013-2015)
- Collaborated to develop software for an autonomous and human-controlled competitive robot
- Won 2014 Pittsburgh regional and advanced to world championship