View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCL Discovery

SIFT: Building an Internet of Safe Things

Chieh-Jan Mike Liang[†], Börje F. Karlsson[†], Nicholas D. Lane[†], Feng Zhao[†],
Junbei Zhang[⋆], Zheyi Pan[‡], Zhao Li[⋆], Yong Yu[‡]

[†]Microsoft Research [⋆]USTC China [‡]Shanghai Jiao Tong University

ABSTRACT

As the number of connected devices explodes, the use scenarios of these devices and data have multiplied. Many of these scenarios, e.g., home automation, require tools beyond data visualizations, to express user intents and to ensure interactions do not cause undesired effects in the physical world. We present SIFT, a safety-centric programming platform for connected devices in IoT environments. First, to simplify programming, users express high-level intents in declarative IoT apps. The system then decides which sensor data and operations should be combined to satisfy the user requirements. Second, to ensure safety and compliance, the system verifies whether conflicts or policy violations can occur within or between apps. Through an office deployment, user studies, and trace analysis using a large-scale dataset from a commercial IoT app authoring platform, we demonstrate the power of SIFT and highlight how it leads to more robust and reliable IoT apps.

1. INTRODUCTION

Through the proliferation of open programmable objects and devices, IoT systems in homes and the workplace are rapidly going mainstream. Projections suggest that, in the next five years, more than 20 billion networked devices will be installed in the environments we work and live [22]. Industry is responding to this new normal with universal protocols and interfaces (e.g., AllJoyn [1], Thread [6]) that allow these devices to inter-operate, and thus provide rich and adaptive user experiences.

Although the adoption of IoT technology is rapidly advancing, technology that ensures these systems are safe and secure is lagging significantly. This is quickly creating a critical problem, in which IoT systems are likely to appear in common usage, but without the verifiable guarantees necessary for users who must live in these connected environments to feel safe. A recent study of the top 10 IoT devices across dozens of categories found that most devices presented a broad range of known vulnerabilities [23], from networking, firmware, authentication, programming/device interaction, to encryption. Specifically, 60% failed to encrypt network traffic (even for device software updates!) and 80% used weak authentication with the cloud. This is even more alarming if we consider IoT devices increasingly have even lethal ties to our physical well-beings (e.g., insulin pumps) and safety [17, 18].

A fundamental piece in the emerging puzzle of IoT safety, security, and privacy is understanding precisely what new breakthroughs in programming support are necessary to enable users to develop and operate safe IoT apps. Although attention has largely focused on vulnerabilities more familiar to networked systems (e.g., authentication, encryption), new *safe* IoT development tools are needed for two reasons. First, programming is central to the customization of an IoT system to the required behavior and preferences of users. More importantly, a bug in an IoT app could have its effect greatly amplified by the physical world. Second, already tens of thousands of early-adopters are using commercial services like IFTTT.com [4] to author IoT apps that automate various parts of their lives. Yet, programming support is still in its infancy – virtually no IoT development tool exists today to help improve key safety aspects of IoT apps.

In this paper, we present SIFT (Safe Internet oF Things), an IoT framework that provides the key programming support for users to customize the behavior of their IoT systems. The goal of SIFT is to radically reduce the user effort needed to produce *safe* IoT apps by introducing a series of automated techniques for verifying key safety criteria and providing the necessary support for users to easily correct problems that are detected. Because of the breadth of safety problems, SIFT defines IoT safety problems by two most important system properties: *verifiability* and *determinism*. First, SIFT ensures apps respect custom policies – for instance, policies describing safety limits ranging from hardware specifications to the CO_2 concentration in a room – and verifies that an IoT app will behave within the bounds specified by the policies. Second, SIFT guarantees apps do not result in logical conflicts or policy violations at runtime. The former captures non-deterministic system states, such as when two apps specified by users attempt to both open and close a door at the same time, or when a faulty IoT device results in unpredictable app behavior. Although SIFT primarily targets end-users (e.g., home-users, employees, IT staff) whom we anticipate will create IoT apps, we also expect those who install, maintain and even build IoT systems will use SIFT to write and verify policies for their devices or environments in which these devices are deployed.

Under SIFT, users program an IoT app with a series of event-based rules. While individual rules remain simple, by aggregating rules together, complex system behavior is possible. This programming model is natural for non-expert end-users to express their intended behaviors for the IoT system. It also allows developers to rapidly build large and rich apps from simple rules. Unlike any current solution, SIFT provides the key support for developing an IoT app

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IPSN'15, April 14 - 16, 2015, Seattle, WA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3475-4/15/04 ...$15.00.
http://dx.doi.org/10.1145/2737095.2737115.

and ensuring its safety properties before being deployed into the wild. First, SIFT uses its *synthesis engine* to generate code that is specific to the deployment environment. Users no longer have to specify, for example, which exact IoT devices in the environment are used to implement the desired behavior – a key source of safety-violating errors. As a result, IoT apps are no longer brittle to devices that become faulty and allow unexpected un-safe behavior to manifest. Next, SIFT verifies this synthesized representation through a *safety engine*. By leveraging recent advances in white-box model checking, SIFT is able to scalably verify that IoT apps do not violate safety policies. Significantly, SIFT allows users of the system (e.g., maintainers or end-users) to specify policies that are considered safe, and prevent any IoT app from executing that breaks such specification. In practice, IoT app development can benefit from either or both of the synthesis and verification support.

The contributions of SIFT are as follows:

- It is a first-of-its-kind development support framework focused on bridging the safety gap plaguing IoT apps today. IoT app safety can now be significantly improved, without increasing efforts from non-expert IoT users. While related model-checking efforts in verifying cyber-physical systems (CPS) exist [10], these efforts typically target domains, e.g., automotive control, where expert-built models are readily available. In contrast, IoT app verification must address imprecisely defined user intents, complex interactions between many requirements and things (possibly deployed over time), and incomplete rule specifications.

- SIFT formulates the verification problem for IoT apps as formal model checking, and provides tools for specifying and efficiently checking of key IoT app safety properties. Safety is tested with respect to both logical conflicts, e.g., conflicting rules, and policy conflicts that enforces, for example, device operating parameters, privacy agreements, or user preferences. We leverage the state-of-the-art program verification techniques for both efficiency and scalability.

- SIFT provides an easy-to-use authoring environment for IoT apps that lessens the development burden and promotes app safety. SIFT is able to synthesize a targeted deployment-specific implementation based on a high-level specification by the user. Not only is this easier for the user, but it also removes important forms of un-safe behavior that might be introduced through human error. SIFT is also capable of learning and proposing new IoT apps by observing the behavior of users and their common routines.

2. BACKGROUND

The need for *programming safety* in IoT environments is underpinned by the rapid adoption of diverse open systems that in turn lack adequate protection from a variety of concerns. We describe a representative and highly popular commercial platform for developing IoT apps along with key examples of safety concerns that are beginning to manifest.

2.1 IoT Programming with IFTTT.com

Thousands of people use IFTTT.com [4] everyday; this website acts as a platform to author, host, execute and share a series of individual rules that encode a simple semantic of: if *event occurs* then *perform action* (referred to as if-this-then-that). Over 140,000 rules have been shared so far. Although the platform is for generic task automation (e.g., if I am tagged in a photo on Facebook, then email me the photo), IFTTT.com includes extensive support for IoT devices (more than one quarter of all data sources and sinks are IoT related). Today, an 8,000 strong community of IFTTT.com users exchange rules with each other, designed to automate various IoT devices they own (e.g., light bulbs, door locks, lawn sprinklers). 34,000 IoT-specific rules have been created so far, and they include: gradually turning on bedroom lights when it is time to wake up; turning off heating when the user falls asleep (detected by IFTTT compatible wearables); or notifying their family when their car passes a landmark nearby their house.

Representative of existing IoT programming models in use, IFTTT has very limited authoring assistance. Users can compose individual rules (or recipes) with a web interface. This manual process requires specifying the exact device(s) that are involved in the rule and related parameters of the action taking place.

2.2 Emerging Safety Concerns

Our work examines key safety concerns exposed in user-automated IoT systems today, both for individual user scenarios and in cases where apps from multiple users interact on the same environment. Three of the most challenging concerns for non-expert users to cope with are:

Manual Conflict Checking. A key way in which the programming of IoT systems can become unsafe is through conflicts. In this context, there are several types of conflicts to consider. First, conflicts can emerge when two or more instructions given to IoT devices cannot be satisfied simultaneously. A simple but practical example of this is when two instructions are provided to a single device simultaneously, both of which can not be executed. For instance, a single light-bulb may have two simple rules provided to it – one that requires it to be turned on during evening hours, and other that requires it to be turned off when no one is in the room. Conflicting IoT programs can occur with a single user who perhaps does not realize instructions can conflict. Or through multiple users who encode opposing preferences. Second, conflicts can arise between an app rule and a predefined policy. For example, turning on a light based on time can violate an energy-conserving policy that turns off light based on room occupancy. In these two cases, what is required are automated methods to highlight to users when such situations arise *before* they become a problem.

Complexity of Programming. A large amount of IoT system utility is only available if users are able to specify how they wish devices to interact and operate. However, available programming models remain too complex to use safely. There are two key sources of complexity. First, fundamentally IoT applications include a significant amount of event-based (for example, context driven) logic that is well known to be error prone to author [29]. This problem does not disappear even if the building blocks of programming model are simple (such as the popular if-then-else semantics of IFTTT) as the logic that needs to be encoded does not change. Second, IoT development today assumes a very low-level abstraction where users are highly aware of *how* device

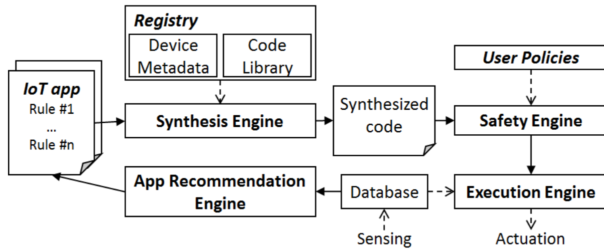


Figure 1: SIFT Architecture.

behavior is performed. For example, users must determine exact source devices of information (such as, the specific door lock), as well as which specific devices will provide any required actuation. Even worse, users must manually specify parameters in the logic they provide. Not only is it hard and error prone for users to specify their IoT applications, but also the resulting IoT apps are brittle and tied to the particular IoT deployment (e.g., which devices used, where are they placed) at the time of authoring. Significant opportunity for more robust and safer IoT app behavior is possible if users are able to have low-level operational details automatically generated from high-level descriptions that are separated from the system deployment.

Lack of Feedback. Available methods of IoT programming today offer few tools for users to examine, diagnose and debug deployed code. Even if problems such as conflict checking are detected, users still require ways of understanding what is causing the conflict and even be offered ways of solving the issue. What is missing are techniques that provide feedback to users in intuitive ways to understand how apps can be made safer.

Collectively, these problem are increasingly exacerbated as the size and complexity of IoT systems built increase. This can happen even in scenarios where system requirements or the programming model may appear to be relatively simple. For example, in home scenarios with IFTTT-like rules, as these systems are used by non-expert users for long periods, the number of rules and their interactions can increase beyond what users can comfortably understand.

3. SIFT OVERVIEW

We begin our description of SIFT with an overview that also examines the considerations that underpin its design.

3.1 Design Goals

The design of SIFT is grounded in the following objectives:

Safer Programming with Low User Effort. Most approaches to IoT programming in current use make strong assumptions as to the user's programming skills and understanding of device behavior. Users need to discover and manipulate individual devices around them – as well as understand how devices should best interact to achieve user objectives. IoT apps are written with a low-level abstraction directly addressing devices. Users are empowered under such assumptions, but this can be dangerous!

As IoT gains mainstream usage, not only is this device-centric paradigm more difficult (and less portable), but it is also exposed to user errors and system failures. To this end, SIFT adopts a semantic programming model that allows non-expert users to focus on their intents rather than which specific device are used to achieve their needs. Data-centric operations are then translated by SIFT into device operations at runtime. Furthermore, this approach lowers the user effort as such suggested programs are already verified IoT apps, rather than those built by hand.

Target Common IoT Safety Violations. Guided by our own user study (see §6) and the findings of various studies, two common safety-related problems are: conflicts among app rules, and violations of policies of various types (potentially written by multiple stakeholders). As mentioned in §2, the challenge is to help developers debug and receive feedback on these potential problems *before* apps are deployed. Examples of the former are: finding conditions where contradictory app rules would be triggered to act on a single connected device or where state of a device becomes unstable as multiple apps change it in a rapid looping sequence. Reporting these violating conditions guides developers in tuning app rules. For the latter, policy verification requires the knowledge of impacts on the physical world due to individual rules. Modeling physical world entities can approximate such impacts and provide programming guidance to users.

Actionable Safety Verifications and Guidance. For the same reason end users require automated verification of their IoT apps to avoid unintended harm, end users also need assistance in understanding how to correct problems when they are detected. It will be unclear to a non-expert why an app has failed without precise feedback. Therefore SIFT should use the same model checking algorithms that help verify when problems appear to also indicate key pieces of information. For example, users should be informed of the type of problem and locations in their IoT app the issue emanates.

The SIFT platform provides this end user experience via the architecture shown in Figure 1. Next, we describe functionalities and challenges addressed by system components in running and verifying IoT apps.

3.2 Safety Promoting Semantic Programming

SIFT allows an IoT app to be constructed as a set of rules defining device triggers and actions. For example, a home owner might write an IoT app to automate light behavior and define a series of rules that describe the times and conditions (such as when a room is occupied) that trigger lights in a room to be turned on and off. We note that several apps can share a common subset of rules. Individual rules can also query for data points of a spatial and temporal instance, and can change a device state. The grammar of the SIFT programming model is as follows. We note that `app_rule_name` enables rule chaining.

```
IF (data_query) THEN (actuation || app_rule_name)
                ELSE (actuation || app_rule_name)

data_query := SELECT (data_type) IN (data_unit)
              AT (location)
              BETWEEN (time_from) AND (time_to)

actuation := UPDATE (device_type)
             AT (location)
             SET (device_state) = (new_state)
```

A SIFT app to control a humidifier based on room occupancy can be written as follows:

```
IF (SELECT people IN int AT living_room) > 0
THEN (UPDATE humidifier AT living_room SET switch = on)
ELSE (UPDATE humidifier AT living_room SET switch = off)
```

Under SIFT, there are two ways for users to create apps. The simplest method is for them to accept (and likely edit) recommended rules that are auto-generated by **App Recommendation Engine** (described in detail in §5.1). However, the primary way users build an app with SIFT is by specifying their own bundles of rules (as above) and relying on the **Synthesis Engine**. Users then submit their app to the SIFT platform, where app rules are first processed by the Synthesis Engine, which builds an execution plan tree to translate user-friendly app rules into machine-friendly instructions.

The Synthesis Engine can handle a diverse set of app types. First, for rules targeting a space (e.g., *temperature of the living room*), it uses Device Metadata Registry to find data sources (upon connecting to SIFT, all devices register the following metadata specified by system maintainers: ID, model number, location, sensing type, and unit).

Second, rules can query for data points that do not exist, possibly due to unit mismatches or even the lack of physical sensors at the points of interest. In these cases, the Synthesis Engine identifies functions in Code Library to transform data or estimate answers (with attached confidence values). Code functions are contributed by users in the ecosystem or by outside experts that have the knowledge on data processing (e.g., counting people in a picture). Forward and backward chaining are two general approaches to select code functions, with the difference being problem solving direction. Backward chaining works from the end goal (i.e., query) and finds a chain of functions (based on pre- and post- conditions) that would lead to a set of devices. Given the potentially large number of connected devices and code functions, SIFT uses backward chaining known to be more efficient when the answer space is too large.

Third, the Synthesis Engine needs to translate semantics into machine-friendly instructions. For example, turning on lamp x can mean sending the string `{"cmd":"2","val":"1"}`. To help build the command mapping for new connected devices, we recommend similar devices previously known in Device Registry for users to update. The recommendation algorithm ranks devices based on location proximity, device model, and sensing type.

The output of Synthesis Engine is an execution plan tree whose nodes represent data sources or code functions to execute. Then, **Execution Engine** executes the synthesized tree, by traversing edges from leaf nodes up to the root.

3.3 Safety Verification

Two categories of safety problems are tested by the **Safety Engine** before an app is executed: conflicts among app rules, and nonconformity with user policies. Policies are written in the form of conditions that should not happen in the physical world (e.g., the house temperature should not be over 40°Celsius). Verifying IoT apps is inherently more difficult than traditional software packages, due to interactions with the physical world. One example is that, while temperature threshold is one common type of policy, checking policies requires at least the knowledge of HVAC's capabilities. The problem becomes more challenging, as we consider cross-domain relationships. In the previous HVAC example, since temperature can influence humidity, how should the Safety Engine check the HVAC-related app (which specified a user's humidity preference) against, for instance, a manufacturer's temperature-defined operating policy?

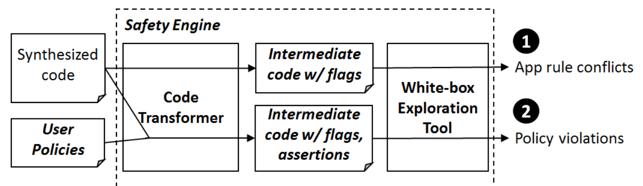


Figure 2: Architecture of Safety Engine, which checks for two types of safety violations: (1) app rule conflicts and (2) policy violations.

The basis of both safety problems can be reduced to satisfiability: Does there exist a condition that would trigger multiple rules to act on a single device, or trigger an user policy violation to happen? While theorem provers seem natural for these two problems, §4 explains our experience with them and motivations to consider other approaches. To this end, SIFT adopts white-box exploration which has been commonly used for testing software problems by automatically feeding input parameters. However, most white-box exploration techniques are not designed for IoT systems (one example: they often assume sequential program execution, a characteristic not present in most IoT apps programming models – such as SIFT and IFTTT.com).

Figure 2 shows how the Safety Engine checks for rule conflicts and policy violations. First, an IoT app is converted (using the Code Transformer) into an intermediate format compatible with existing verification tools (described as the White-box Exploration Tool). This process also includes important annotations with flags and assertions that capture the true semantics of the IoT app. For code exploration tools, an assertion triggered would signal an exception. In the case of SIFT, assertions check for invalid device states and policy conditions.

4. SAFETY ENGINE

In what follows, we detail the design of the SIFT Safety Engine. Later in §5.2, we describe how this design is implemented with state-of-the-art model checking solvers and symbolic execution.

Is Theorem Proving Alone Enough? The goal of software verification is to prove properties of programs; tested properties range from simple assertions, which state that a condition holds when a particular code location is reached, to global invariants, which check that certain predicates hold on every reachable state. In the case of app rules under SIFT (see earlier example in §3.2), similarities between the rule structure[1] and logical first order formulas suggest the use of automated theorem proving technologies as a *seemingly* suitable approach for verification.

The conventional approach in a verifier architecture is to convert the source language under analysis to an intermediate representation that preserves its characteristics and

[1]Incidentally, IFTTT-like rules as well.

facilitates reasoning over it. Once rules are translated to a logical representation that models the constraints over its triggers and actions, the satisfiability of these constraints can be checked by a constraint solver. However, doing this for IoT apps is problematic. The translation itself is not simple, for reasons that include: (1) rules can be chained; (2) devices might have multiple non-boolean states; (3) there is no a priori implied precedence between rules; and (4) rules frequently make use of more than just boolean logic (like arithmetic). Moreover, other capabilities than just identifying logical conflicts are needed, such as: checking of additional safety policies, violation of user preferences, and identification of specific situations and inputs that lead to problems.

To address these challenges, our design utilizes a three-stage approach combining the use of a SMT-solver and run-time code exploration. Formally, we consider safety verification problems specified in the following form: the input is a set of IoT rules (τ) – i.e., an IoT app – and a set of policies (φ), and the output is *safe* if τ is deemed satisfiable and no reachable execution path reaches a state that violates φ or any other τ, IoT app. We now describe each Safety Engine stage in turn.

4.1 Detect IoT App Rule Conflicts

As discussed, verification or analysis tools require a model of the IoT apps being analyzed that is amenable to symbolic reasoning (i.e. well-formed formal expressions that allow valid deductions). Thus the Safety Engine begins by parsing rules of τ into a set of logical rules (the model) representing the system input, that can then be verified for app rule conflicts through the use of symbolic execution testing. App rule conflicts occur, for example, when two rules within two IoT apps try to cause different (and incompatible) actions on the same device simultaneously.

Next, a SMT solver is applied to this intermediate representation (model). Satisfiability modulo theories (SMT) generalizes boolean satisfiability (SAT) by adding support for other relevant theories (e.g. arithmetic, arrays, quantifiers) [19]. The SMT solver decides the satisfiability of the generated logical formulas. If the model is deemed unsatisfiable, an *unsat core* is produced containing the conflicting rules. (More details provided §5.2). It is important to note that it is not possible to guarantee the unsat core will be minimal, so extra processing to weed out rules not involved in conflicts might be needed.

If the model is satisfiable, then this means solutions for it indeed exist. However, specific situations (i.e., rule parameter values) might still lead to conflicting actions. As a simple example, the expression: $(T >= 25 \implies AC = true) \wedge (T <= 25 \implies AC = false)$ is satisfiable for any value of T different than 25. But for 25, a conflict exists. Coping with this situation could be accomplished with a more refined intermediate representation (model). However, recovering the reasons for the conflict (i..e, which rules were involved and exact parameter values that resulted in the conflict) from the unsat proof might be impractical. This can occur during skolemization (often SNF) and simplification steps – often when optimizing the search for proofs may produce expressions not necessarily equivalent to the original ones (but still equisatisfiable).

Finally, to discover which specific instances of environmental inputs (e.g. sensed values) can trigger conflicts, we

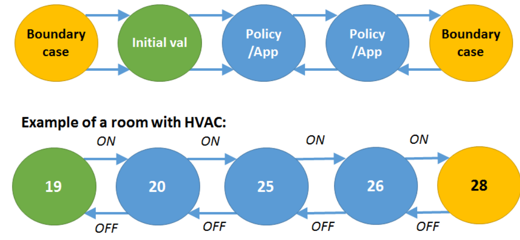


Figure 3: A state diagram models an HVAC maintaining the room temperature between 20 and 26°Celsius. The user also specifies an negative policy for temperature reaching 25°Celsius.

combine model checking with symbolic execution. Symbolic execution techniques enumerate all program paths, and generate and solve constraints for each enumerated path. Essentially, this step works by repeatedly executing the rule set (IoT apps) and solving constraint systems to obtain inputs that will guide the search towards conflict-triggering execution paths. The problematic generated inputs being the situations we want to identify. This also allows us to easily check for additional properties like safety policies and environmental models (e.g., through the use of properly placed assertions in the executed code). We consider these properties in § 4.2 and §4.3.

4.2 Identify Policy Violations

Policies encode real world conditions that users do not want. For example, `policy((SELECT light_tsr IN lux AT balcony BETWEEN 7AM and 6PM) == true)` specifies that the balcony light should not be ON during the day. SIFT analyzes app rules during submission to catch potential violations as part of the development process. To this end, our design extends the symbolic execution tool for catching app rule conflicts described in the previous subsection.

Policy checking begins with the device and real world phenomenon states that need to be tracked. In the example above, the balcony light is such a state to be tracked. Likewise, location information of individual app rules suggest the scope of the actuation. Next, user policies are translated into a series of assertions that are inserted at the end of each rule. This forms the policy-version of the intermediate language required for violation testing. By again applying symbolic execution techniques, the Safety Engine can enumerate all program paths attempting to solve all inputs. In other words, they look for any possibility to make an assertion true. If any assertion is evaluated to be true, the Safety Engine reports this exception to users before triggering the app. Because it is clear which assertion became true, users can be informed not only which rule of their IoT app breaks which policy, but also which environment trigger or actions (and even the parameter range if relevant) – such actionable feedback assists users in correcting violations.

With the changes described so far, the Safety Engine can verify policy violations related to discrete states. However, policies can also concern real world phenomenon which has continuous state (e.g., room temperatures fluctuation within a range of values). The challenge is to capture and model these phenomenon during code analysis. Our solution is to construct a state diagram that captures transitions among critical states from the continuous space. Critical states in-

clude the following: (1) the initial/default value, (2) min and max values possible (according to device specifications), (3) values specified by the condition of each policy. Figure 3 provides an illustrative toy example of this process with a room and an HVAC. In this case, state transitions are created according to device specifications. As shown, states are associated with the policies they violate – thus when the Safety Engine is inspecting an IoT app, a violation is flagged if any of reachable states has a policy violation.

4.3 Coping with Cross-Domain Variables

Verifying policies simply by inspecting the triggers, actions and parameters explicitly described in the policies *themselves* and IoT apps is brittle. For instance, a policy might detail the maximum level of CO_2 in a building floor, but a user IoT app might only refer to opening windows, and disabling air conditioning – which also may result in unacceptable CO_2 levels, and therefore is unsafe. Enabling verification of latent consequences of IoT apps requires some understanding of how cross-domain variables relate to each other.

Clearly, completely solving this problem is unrealistic as it requires deep comprehensive environmental and physical models. For this reason, SIFT adopts a conservative (but practical) approach relying on readily available sensor data to minimize this concern in a tractable manner. Our design centers on a datastore of cross-domain correlations to test for violations when entities are not of the same type. In the previous example, the datastore would contain a simple model for estimating the CO_2 in-flow to the building based on the number of open windows and duration of time the air conditioning (that provides air cleaning) is disabled.

In addition to be populated by system installers and maintainers, the datastore can be dynamically constructed by estimating the relationship between IoT device actions/triggers and environmental condition of interest (e.g., temperature, light). The design of SIFT is agnostic to any specific modeling technique – currently, we build a Bayesian network based model that incrementally learns from data and traces of IoT usage. Over time the system can learn how this positive relationship is parametrized within a particular building deployment – note, unlike many cases of learning where manually labeled data is required, SIFT can perform this modeling without any supervision (i.e., no user involvement). During the verification, SIFT changes modeled variables (e.g., CO_2 level) with actions of app rules (e.g., opening a window).

5. IMPLEMENTATION

SIFT is implemented as a series of independent modules written in C++ and Python that co-ordinate via a TCP-based control plane. In sum, our SIFT prototype spans 8,950 lines of code and primarily resides on cloud servers with end-devices connecting via an open RESTful interface. Our design allows SIFT to scale as required, by dynamically allocating additional instances of each module based on traffic demand. The exception to this being the centralized component for model checking found in the Safety Engine (see Figure 1) that can not easily be parallelized.

In the remainder of this section, we focus on the two core SIFT components providing semantic programming and safety verification.

5.1 Semantic Programming Engines

Three programming engines are built into SIFT, and the operation of each in our prototype is described below.

Synthesis Engine. For each IoT app rule, the Synthesis Engine extracts the trigger or action clause. Those clauses with queries that no device can directly satisfy, the Synthesis Engine uses a backward chaining algorithm to find REST calls offered by IoT devices that either return (or estimate, transform at a minimum) the desired result. Importantly, SIFT allows REST calls to express a confidence value in their result returned (key for those that estimate their response). Also while REST calls are often device based they can also correspond to Code functions contained in the Code Library.

Both the Device Metadata Registry and Code Library Registry are built using the SICStus Prolog engine [32]. Backward chaining is parameterized by matching pre- and post-conditions of IoT app rules. This process can also be done incrementally, by adding one component to the chain of rules at a time; the stopping condition is when a device is included in the chaining output. Some REST calls may require more than one input. In these cases, the output execution plan is a tree that links the output of multiple iterations of backward chaining. These execution plans are consumed by the Execution Engine.

Execution Engine. Runtime usage of an execution plan is relatively simple. The Execution Engine begins by traversing the tree from leaf nodes to root. Depending on the specification of each node in the tree, the Execution Engine performs a different operation, which includes: collecting data from a IoT device (or historical store of the information), or making a RESTful call to a component (or IoT device) within SIFT, for example, to perform some actuation.

App Recommendation Engine. Based on patterns of manual (user initiated) usage of IoT-enabled devices, this engine generates IoT apps that non-expert users might not be able to author correctly (e.g., missing conditions). To accomplish this, we adopt and modify association rule mining techniques [12]. At a high-level, this approach searches the combinations of concurrent IoT device activations (e.g., events) and selects candidate combinations ($\{X, Y\} \implies Z$) – for example, lock all doors when leaving the house, *if* you are last person in the home) – based on computed values of *support* ($\phi \cdot \frac{\sigma(\{X,Y\} \cup Z)}{N}$) and *confidence* ($\frac{\sigma(\{X,Y\} \cup Z)}{\sigma(\{X,Y\})}$). We use a relaxed notion of concurrent activation and treat multiple activations within a 15-minute window (θ) as concurrent (θ tuned as required). Our design deals with two additional challenges. First, we correlate only events happening within a proximity or space. This helps both reduce the search space for the algorithm, and prevent recommending unreasonable app rules that tie two events happening far away. Second, we scale the support of candidates (ϕ) based on how likely an app would be useful to users. Our current implementation leverages a large open repository of common sense (ConceptNet3 [2]) that provides a quantification of how strongly related multiple concepts typically are; for example, concepts like “leaving a house” and “locking doors”. As a result, if candidates are composed of unrelated device activations it is penalized and less likely to recommended. Tackling this issue is important because standard association rule mining is prone to generating unnecessary rules.

5.2 Safety Engine

The core of the Safety Engine is a model verification pipeline. Provided rules from IoT apps act as pipeline input, and are translated into an intermediate format where specific conditions can be verified. SIFT takes a pragmatic two stage process to analyze IoT apps (rule collections) and test for both App Conflicts (§4.1) and Policy Violations (§4.2).

Stage One – Logic Inspection. Safety checking begins with a basic series of logic checks that act as a sanity test on a tree representation of the IoT app rules. The goal is to analyze for conflicts caused by triggering of contradictory or incompatible actions. These can be either two actions on the same device (e.g. turning on and off a light) or opposing actions by two devices (e.g. turning both AC and heater on at the same time). This process takes advantage of the structure of the rules in that they already resemble logic formulas ($A \implies B$) and so we translate them into a combination of formulas to be checked for satisfiability, encoded following the SMT-LIB standard [8]. The resulting set of formulas is then checked using the theorem prover and constraint solver Z3 [19]. If Z3 deems the model unsatisfiable we retrieve an unsatisfiable core from Z3 that can identify which rules (and which specific values) lead to the conflicts. This valuable feedback is provided back to the end-user.

However, as we learned in §4, other types of safety need to be tested by SIFT. As shown in [9], software model checking can be used to find program paths satisfying certain coverage conditions (in our case, a path reaching a particular conflict by triggering particular rules), and the symbolic constraints generated from the path are solved by a constraint solver to produce test inputs (i.e. the environment situation that would trigger the problem). Symbolic execution (a technique similar to concrete execution, but where symbolic variables are used for the program inputs) can be used to obtain these constraints. Such constraints are then combined with additional ones to eliminate situations that could never happen in the environment where the IoT apps run, which allows identifying problematic inputs.

Stage Two – Symbolical Execution. A core challenge in the symbolic execution for SIFT is in the automatic generation of suitable intermediate representation for use by conventional model checking algorithms. In particular, rule collections must be transformed into actual code that preserves the characteristics of the execution environment of the IoT apps. In-line with current programming models (such as IFTTT for instance), SIFT does not have an implied precedence between rules – all of them are active at a given time and in this way they do not behave like more familiar firewall rules or e-mail filters (situations where rule order matters). A further complication is that users are allowed to chain rule activations together within IoT apps. For example, a user may enter a house and turn a light on which is tied to turning on the air conditioner – an intermediate representation must account for this additional dimension. The Safety Engine adopts a bounded model checking flavor of solution. It begins by unrolling the control flow graph of the IoT app for a fixed number steps (k). Conceptually checking for unsafe situations extends within this number of k steps. To perform this, C# code is generated where each rule becomes multiple statements including an IF statement along with a set of supporting additional ones. Statements sit within a loop structure iterating k times. This design provides equivalence to chaining without precedence whereby statements mapping to a rule can fire each state.

To illustrate this intermediate representation we show an example with two rules that is chained three times:

```
WHILE (k < 3) {
  IF (tr1 AND !f1) THEN (action = A AND tr2 = True AND
                         f1 = True)
  IF (tr2 AND !f2) THEN (action = B AND f2 = True)
}
```

(Clearly, A and B can not happen simultaneously in this example intermediate code.)

Using this representation we can integrate Pex into SIFT. Pex is a automated white box testing tool for .NET [33]. It performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along all execution paths. If a conflicting situation is triggered we can then retrieve the input parameters that led to it, as all as the triggered rules (flags). Through Pex we can perform the App Conflict and Policy Verifications described in §4. The state machine used for policy checking is implemented as a separate module within C# and is easily integrated into the Pex operation as it is designed to be extended with such modules. When necessary for policy checking, the datastore for cross-domain data types are accessed, each of these stored models is also written in C# for ease of operation. Periodic generation of models by running the Bayesian network across the accumulated environment and IoT device data occurs – by default this interval is 1-week.

6. EVALUATION

We present our results from an office deployment and a large-scale trace analysis with data collected from IFTTT.com. The key findings of our experiments are as follows:

- Even for users with technical backgrounds, unintended conflicts (both intra-app and inter-app in multi-user settings) occur at a surprising rate. SIFT is able to detect and report such conflicts in a timely fashion. Furthermore, conflict detection remains responsive for rule set sizes (i.e., collections of IoT apps) typically created by users today.
- The Synthesis Engine is able to service the IoT apps that users specify and auto-generate deployment/device-specific code that meet the users' requirements. We observe no example where: (1) the synthesized code is not in-line with the user's expectation; or, (2) users are unable to express their requirements.
- Policies are created by users for a number of situations. In our experience, we notice they are inventive and embrace the approach. Further, we find that execution time at the level of usage seen during the deployment is sufficient, and not a bottleneck.

6.1 Methodology and Experiment Setup

Our evaluation includes both a small-scale office deployment and home-oriented data-driven simulations. While the deployment provides a real-world environment for experiments, simulations provide a way to study larger-scale user and system behaviors.

	User #1, #2	User #3
Light sensor	√ Env brightness	√ Env brightness
Temp sensor	√	√
Humidity sensor	√	√
Motion detector		√ Seat occupancy
Air quality sensor		√
Pwr outlet switch	√ Lamp	√ Lamp

Table 1: Devices in office deployment.

	01	02	03	04	05	06	07	08	09	10
Re-used	35	33	25	25	25	19	30	10	10	30
New	12	4	6	21	2	11	11	10	16	3
Include chaining	11	4	6	16	2	7	6	5	8	3
Individual parts	123	64	75	97	38	123	74	66	42	51

Table 2: Distribution of app rules (ordered by ID), as chosen by users for home automation.

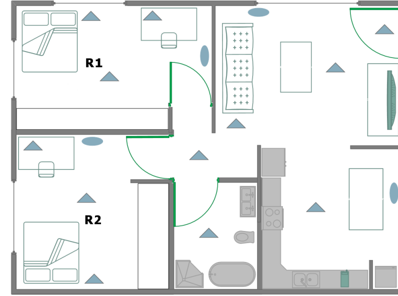


Figure 4: Floor plan of a "standard house", used in the user study to elicit app rule sets.

Office-oriented Small-scale Deployments. To study how users and SIFT interact with a diverse set of devices, we instrument the working area of a floor inside an office building with connected devices. Table 1 lists our deployed devices that are connected to the SIFT back-end. All sensors report new readings every three seconds. We report results based on three weeks of data.

Home-oriented Data-driven Simulations. In order to evaluate the safety engine, we need access to (1) IoT apps that when executed together (i.e., simultaneously) by users might result in conflicts and (2) deployment environments. As a repository of such apps is currently not available and deployments would vary in structure and availability of device types, we need to synthesize IoT app workloads.

IFTTT.com is currently the service that comes closest to representing a large scale repository of IoT apps. As such, we crawled all IoT-related *recipes* available online (i.e. rules shared by users of the service). We then analyze this dataset, focusing on the popularity of rules and the distribution of devices involved, to come up with representative devices and automation behaviour for a home-oriented scenario.

However, this still does not address a number of missing deployment details (device placement, for example). We gather these missing information via a user study. To facilitate this we use a standard floor plan that interviewed users could treat as the target deployment environment when defining their own IoT apps. The floorplan of this standard house, and the location of devices in it, are shown in Figure 4. To assist these users in creating more diverse apps and leveraging available devices, we also provide them with generalizations of the most popular 35 recipes (related to IoT) available from the IFTTT.com service. This has also the added benefit that it allows us to collect parameters for these popular recipes from real users.

We then performed a user study with 10 unrelated users, with technical backgrounds, but no previous IoT automation experience. Users were asked to choose a room as their supposed bedroom in the standard floor plan and what IoT apps they would like to have active in such environment (both apps active in their bedroom and in the whole house).

Interviewed users were instructed to either pick from the examples apps provided and fill their missing parameters, or to come up with brand-new apps (which could be completely different and even possibly include extra devices). It is important to note that rules from IFTTT.com only allow the simple if-this-then-else structure; not allowing chaining of rules nor multiple triggers and action on the same rule (e.g. using AND). Table 2 shows the distribution of apps created chosen by the users in the study. One can see that multiple users indeed created their own apps, and some of them made use of the added programming model flexibility.

6.2 Programming Synthesis

Our office deployments executed SIFT apps both manually authored by users and automatically recommended by the system. We highlight interesting case studies below.

6.2.1 User Apps

Users in our office deployments wrote several apps to help them better adapt to environment dynamics. These apps are the natural choice for users with no prior experience with IoT-enabled devices. One interesting observation is that sharing code functions (stored in Code Library) can guide users in writing apps. For example, users initially focused on simple apps that read physical sensor data such as temperature and room brightness. Then, we added the code function, `ApparentTemperature`, that calculates the human perceived temperature by considering the humidity. This inspired User #1 to write `ComfortIndex` to estimate his environment comfort. Since people define comfort differently, User #3 copied the code function and customized it.

Figure 5 shows the synthesized execution plan of the apps of User #1 and #3 described above. The figure shows component reuse between two execution plan trees. In addition, both trees have multiple layers that represent chains of code functions and devices.

6.2.2 App Rule Recommendation

We illustrate the usefulness of app rule recommendation, with an interesting case from User #3. After collecting two days of data from User #3, AppRecEng recommended the automation app below for turning on and off a lamp. In fact, this recommendation considers occupancy, which is a condition that User #3 missed if he were to author the rule himself. This observation highlights the value of recommendations in assisting users authoring "correct" IoT apps.

```
IF (SELECT env_brightness IN lux AT 12435) <= 29.907
   AND (SELECT occupancy IN boolean AT 12435) == true
THEN (UPDATE light AT 12435 SET state = on)
```

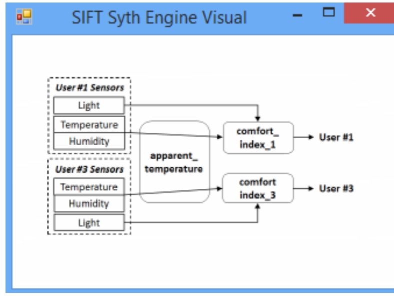


Figure 5: Two multi-level execution plan trees synthesized from IoT apps of office deployments.

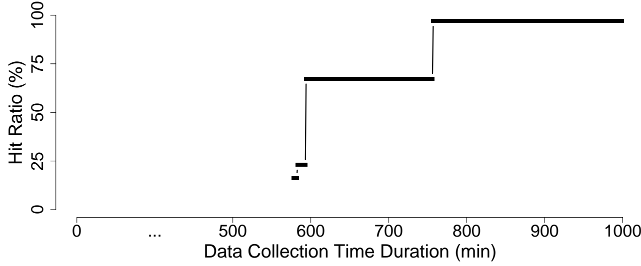
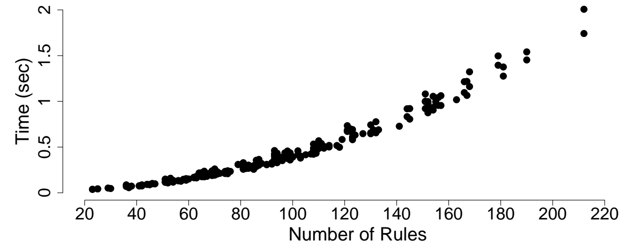


Figure 6: The hit ratio for app recommendation increases with the amount of data collected for training. This result specifically shows this behavior manifesting for User #3's light control.

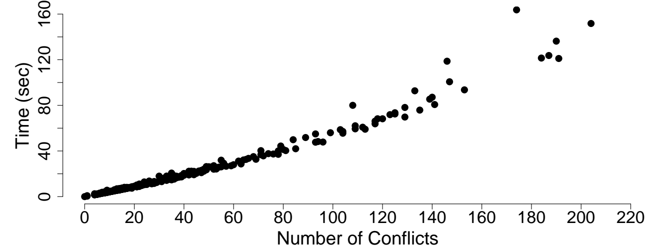
We note that the confidence of a recommendation largely depends on the frequency of the recurring pattern (r) and the number of counter examples observed (c). Figure 6 illustrates this issue with User #3's recommended app. Each time instance on the x-axis represents the amount of training data from User #3 sensors, which AppRecEng uses to build rules to recommend. Then, we use another set of 500-min data to evaluate how well the recommended rule aligns with real user behavior. First, App Recommendation had enough data (578 minutes) since the data collection started. This was around 5 PM, when the office starts to get dark. User #3 then starts to interact with the desk lamp. Second, as User #3 exhibits more interactions, e.g., turning off the lamp during dinner and before leaving work, App Recommendation had more data points to adjust the rule. Finally, around 13 hours after the experiment started, the hit ratio reached 99%.

6.3 App Conflict Detection

The dataset of apps collected during the user study provides us with enough information on different ways the same home could be automated via IoT apps. With these sets of apps in hand, we can then look at the number of conflicts that they may cause during runtime. Figure 8 shows the number of conflicts detected between each set of apps, both if we look at individual users, or when two users share the same home environment (as mentioned before, we assign users to the two bedrooms in the standard house). The number of conflicts (and what circumstances would trigger them) can not be directly used as a final metric to determine the usefulness of the system, as this output would be the initial feedback from SIFT to users; that could then used to refine the rules in an iterative process. But analysis numbers show



(a) Time to detect conflicts for a rule set



(b) Time to identify parameters that cause conflict, given a number of conflicts to be identified

Figure 7: Timing analysis of the app rule conflict checking mechanism.

that, even for users with technical backgrounds, unforeseen interactions between IoT apps manifest easily (especially in multi-user situations).

To illustrate performance of the safety engine, we ran a series of micro-benchmarks on the two steps of the verification process. Figure 7 shows the time it takes to analyze each app set and detect conflicts. Starting with users individually and combining them 2×2 (when appropriate, depending on their assigned rooms). We can see that the time needed for the analysis increases with rule set size, but even to analyze ≈200 rules, it takes less than 2 seconds.

Identifying precisely the circumstances that lead to each conflict potentially takes much longer during symbolic execution. For instance, exploring 200 conflicts takes around 2.58 minutes if analyzed in sequence (Figure 7). This time could be greatly reduced if the analysis is parallelized, but even using the simpler approach, it is still within expectation as validation would happen at ruleset modification time.

6.4 Policy Violation Detection

Since there are very few reports on verifying policy conformance of an IoT app, this section presents our observations and results on (1) how people think about policies and (2) the kinds of violations that the Safety Engine identified.

For the user study, we observe ∼89% of policies authored by users are related to making the environment safer to occupants. Interestingly, many of these policies have an occupancy component, e.g., "Rice cooker cannot work when there is no one at home" and "Room 1 temperature should never be > 25°Celsius when someone is in the room". This suggests that an efficient way of providing occupancy or location information plays a major role towards safe IoT apps. In addition, there are many policies looking at safety from the perspective of human health, e.g., "Never set TV speaker to more than 90dB" and "Speakers never on when I sleep". During interviews, authors of these two rules explained their

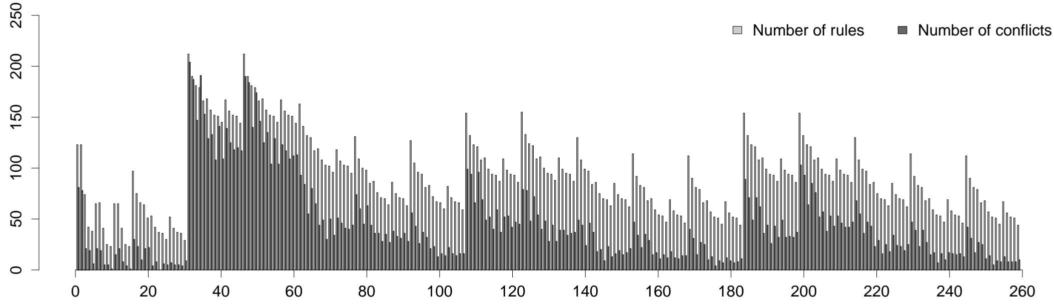


Figure 8: Number of conflicts in different simulations of app combinations.

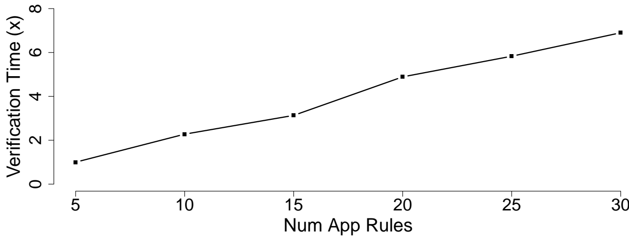


Figure 9: Additional time to verify policies grows w.r.t. number of app rules. Baseline is five rules.

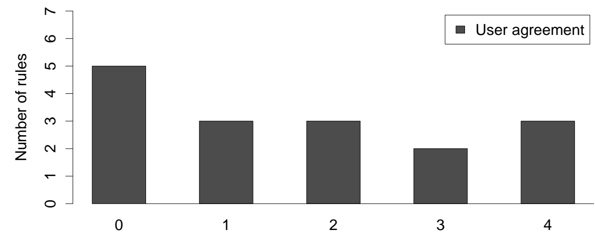


Figure 10: User rating of mined rules.

intention to prevent hearing damage and promote better sleeping environment. We see this as the opportunity for incorporating human-centric sensors to improve safety.

Safety Engine verifies policy conformance of an app before the Execution Engine executes it. We use an interesting case study of User #3 from office deployment to illustrate the usefulness of policy violation detection. Section 6.2.2 presents an app rule that App Recommendation Engine recommended to User #3 for automating lamp control. However, since User #3 is using an old lamp that can get hot enough to be a potential concern, he authored a policy that mandates the lamp should be OFF when there is no one around. Safety Engine was able to identify this potential policy violation, and User #3 subsequently added the following rule.

```
IF (SELECT occupancy IN boolean AT 12435) == false
THEN (UPDATE light AT 12435 SET state = off)
```

A related question is the time required to perform policy verifications. Micro-benchmarks suggest that most of the verification time is taken by the symbolic execution tool to solve constraints and explore all program paths. Since the number of paths is related to the number of IoT app rules to check, we measured the verification time needed with respect to the number of app rules. We artificially generated app rules with random inputs. Figure 9 shows the verification time increases almost linearly with the number of app rules.

7. DISCUSSION

We discuss overarching issues related to SIFT.

Modeling Environment with High Fidelity. Through connected devices, IoT apps actuate and interact with their environment. The type and specification of connected devices and real world entities determine both the scope and impact of their interactions. Modeling the real world is a useful approach for improving safety verification completeness. For example, automotive manufacturers have long been evaluating safety under different real world models. However, the complexity of modeling IoT systems in the real world can quickly explode, considering device type diversity, temporal behavior, physical space layout and so on. As a result, SIFT does not fully model the physical world. In the future, we anticipate exploring the relationship between modeling fidelity and verification completeness as well as approaches towards increasing modeling fidelity.

Run-time Policy Violation Detection. Like most software testing tools, there are types of violations that SIFT cannot identify with code analysis. Some of these undetected violations are due to missing knowledge of how variables can correlate in the real world. For example, opening windows may increase the level of air pollutants present indoors, which can then raise a policy violation. Another example is when IoT devices are deployed in a previously unknown environment. To cope with these cases involving deep knowledge of the real world, SIFT currently also performs run-time policy violation detection on incoming sensor data streams. SIFT captures violations that it understands based on a device and IoT app model; as well as relationships between the system and the real-world that can be learned on-line.

Model Limitations. At this time, we adapt existing model checking techniques that are well understood to verify logical conflicts and policy violations. However, some aspects of rules and the environment are not modeled in the current implementation, such as time intervals and time delays, and interference of device effects in environment characteristics not covered; which can lead to false positives. Examples include: if a light is turned on for only 5 minutes, if a heater

takes 15 minutes to warm up, or if opening the window lets dust come in into the home. We expect, in the near future, to revise our safety engine design to incorporate new techniques (e.g. [11]) that better capture these types of effects.

Actionable Feedback for Non-expert Users. When either a problem of rule conflicts or policy violations is found, SIFT currently points users to offending app rules along with model parameters. While this information has been shown to be useful, non-expert users might need further assistance in fixing offending app rules. To this end, we are currently working on algorithms that take the first step in automatically suggesting concrete fixes for such IoT app rules.

Connecting New IoT Devices to SIFT. There are many efforts on standardizing the communication and management of heterogeneous IoT devices [21]. SIFT leverages these existing efforts to simplify the process of connecting new devices. We take a gateway-oriented approach, where most of our devices indirectly connect to SIFT through a intermediate network device. Specifically, this gateway has the necessary radio and networking stack to communicate with nearby devices with such interaction occurring using RESTful interfaces. By delegating device management to the gateway, we can simplify the architecture of SIFT and operate only against a device abstraction.

8. RELATED WORK

We now discuss how SIFT compares to, and extends beyond, existing IoT related research.

IoT Security Testing. Most current industrial and academic efforts on IoT target connectivity and communication issues, which, from the safety and security point of view, naturally translate into a fixation on protocol design, encryption, and authentication [1, 6]. There is also scrutiny regarding privacy [27]. However, the gamut of issues that can impact users is much broader. SIFT views programming as one building-block activity in practical IoT systems today, and assists non-expert users in addressing problems from imprecise app rules and models.

Depending on the app model considered, verification might not be possible resulting in other approaches to test IoT apps being necessary. Two directions that appear promising are: (1) the synthesizing of inputs received by apps from characteristics of the environment where they run (e.g., [15]); or, (2) automatically extrapolating new test cases based on some templates and information of the IoT apps [36].

Conflicts Detection and Resolution. While event-based interfaces do provide a flexible programming interface and a good degree of usability, most proposed systems do not handle problems that emerge from the interaction among IoT app rules. Rules in general can be hard to debug when the system is not working as users expect; often early adopters simply live with problems or even just turn the rules off [14].

To resolve app conflicts, HomeOS [21] allows priorities to be assigned to IoT apps. However, priorities are not sufficient if two apps have approximately the same importance to users. There have also been efforts on running concurrent applications in sensor networks [13, 26, 38], but these are either limited in resolving conflicts (for example, by imposing overly strong restrictions) or too complex to be comprehensible for non-expert end users. Striking the balance between flexibility of device orchestration and minimizing possible safety violations is key.

Notably, DepSys [28] describes an approach to specify, detect, and resolve conflicts among home IoT apps. DepSys is able to identify dependency issues and resolve them (sometimes involving user intervention). However, it places the burden of properly specifying app intents and dependencies on app developers. SIFT employs a more flexible app model, where a user can author apps, and the synthesis engine is responsible for fulfilling data and control dependencies (e.g., potentially re-using other previously available app parts). Moreover, the safety engine can precisely point out which circumstances cause a particular conflict so users can take action. On the other hand, DepSys lacks the visibility into app internals to make this possible.

CPS System Verification. One problem that the CPS community faces is verifying specifications of complex real-time systems, e.g., automotive control. ISO 26262 [39] (or the more general IEC 61508) is an example of functional safety standards for experts of such systems. Model checking is a common technique in CPS, to automatically verify interactions among system processes and physical environment components. Henzinger et al. [9] introduced hybrid automaton to precisely describe these interactions by combining finite state machines and sets of ordinary differential equations. Bu et al. [11] later tried to lower the verification overhead from having precise hybrid models. While model checkers typically stop when one counter-example (w.r.t. a specification) is found, recent efforts [10] tried to efficiently discover all offending locations in precisely specified models.

However, in the case of IoT where users are typically non-expert, precise and complete models and user intents can be difficult to obtain. The main challenge lies in formulating the verification problem for IoT apps in terms of formal model checking. SIFT provides tools for easily specifying and efficiently checking key IoT app safety properties: app rule conflicts and policy violations.

Declarative Queries for Sensor Networks. TinyDB [24] views a sensor network as a declarative database, and Declarative Sensor Network [16] is a declarative language, compiler and runtime for programming a range of sensornet applications. Both TinyDB and DSN require users to have some level of knowledge of the underlying physical infrastructure and data stream properties. Semantic Streams [37] allows users to issue queries over semantic values without concerning raw sensor data streams or operations. However, SIFT offers a more complete development support with tools for recommending and verifying app rules.

Automation and Ease of Use. While different commercial approaches to orchestrating devices have recently gained popularity [3, 5, 7], these systems behave as black boxes and do not integrate well with devices from other manufacturers. Complex interfaces to configure device interactions and the lack of integration flexibility can present significant barriers for users to fully utilize the potential of IoT devices [14].

In making the programming of cross-device interactions more approachable, research suggests rule-based paradigms (e.g. trigger-action pairs) as a very natural mental model [20] that is also surprisingly flexible and powerful [30, 31]. More recently, Blase et al. [34] examined trigger-action program-

ming in the form of IFTTT-like structures, from a user point of view. To target non-expert users, SIFT builds on these findings and adopts an IFTTT-style programming model.

9. CONCLUSION

In this paper, we study a fundamental piece in the emerging puzzle of IoT safety, security, and privacy: understanding the key programming support necessary for non-expert users to build safe IoT apps. We present a first-of-its-kind IoT development support platform – SIFT. Under SIFT, the effort and attention from non-expert users in producing safe IoT apps are lowered with automated techniques for verifying two key safety criteria: app rule conflicts and policy violations. SIFT formulates the verification problem of IoT apps as formal model checking, enabling the efficient checking of these safety properties. Given the demonstrated usefulness and power of SIFT, we plan to next address user-focused capabilities, e.g., assistance for fixing imprecisely defined user intents and for handling complex interactions between large-scale collections of user requirements and devices.

10. REFERENCES

[1] AllJoyn. https://www.alljoyn.org/.

[2] ConcepNet 5. http://conceptnet5.media.mit.edu/.

[3] Home Automation Systems. HomeSeer. http://www.homeseer.com/.

[4] IFTTT - Put the Internet to Work for You. http://ifttt.com.

[5] SmartThings. http://www.smartthings.com/.

[6] Thread. http://threadgroup.org/.

[7] Wemo. http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation/.

[8] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. http://www.SMT-LIB.org.

[9] T. A. Henzinger. The Theory of Hybrid Automata. In *Proc. of the 11th Symposium on Logic in Computer Science (LICS)*, pp 278–292. IEEE, 1996.

[10] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *Proceedings of (ICSE)*, pp 326–335. ACM, 2004.

[11] L. Bu, et al. Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. In *ACM SIGBED Review 8 (2)*, pp 7-10. 2011.

[12] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[13] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys*, pp 187–200. ACM, 2003.

[14] A. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home automation in the wild: challenges and opportunities. In *ACM CHI*, pp 2115–2124. ACM, 2011.

[15] C.-J. M. Liang et al. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *Mobicom*, pp 519–530. ACM, 2014.

[16] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, pp 175–188. ACM, 2007.

[17] CNET. Car hacking code released at Defcon. http://www.cnet.com/news/car-hacking-code-released-at-defcon/, 2013.

[18] CNN. Why it's so easy to hack your home. http://www.cnn.com/2013/08/14/opinion/schneier-hacking-baby-monitor/, 2013.

[19] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp 337–340. Springer Berlin Heidelberg, 2008.

[20] A. K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. In *Pervasive Computing*, pp 254–271. Springer, 2006.

[21] C. Dixon, R. Mahajan, S. Agarwal, A. B. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, pp 337–352, 2012.

[22] Gartner. Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. http://www.gartner.com/newsroom/id/2636073.

[23] HP. Internet of Things Security: State of the Union, 2014. http://www.hp.com/go/fortifyresearch/iot.

[24] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[25] T. L. M. Kasteren, G. Englebienne, B. J. A. Krøse. Human Activity Recognition from Wireless Sensor Network Data: Benchmark and Software. In *Activity Recognition in Pervasive Intelligent Environments*, Vol. 4 (2011), 165-186.

[26] P. J. Marrón, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. Tinycubus: a flexible and adaptive framework sensor networks. In *EWSN*, pp 278–289. IEEE, 2005.

[27] C. M. Medaglia and A. Serbanati. An overview of privacy and security issues in the internet of things. In *The Internet of Things*, pp 389–395. Springer, 2010.

[28] S. Munir and J. A. Stankovic. DepSys: Dependency Aware integration of Cyber-Physical Systems for Smart Homes. In *ICCPS*. ACM, 2014.

[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pp 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[30] M. W. Newman, A. Elliott, and T. F. Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *Pervasive Computing*, pp 213–227. Springer, 2008.

[31] J. F. Pane, C. Ratanamahatana, B. A. Myers, et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, 2001.

[32] SICStus. SICStus Prolog. Technical report, 2014. https://sicstus.sics.se/.

[33] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of Tests and Proofs (TAP'08)*. Springer Verlag, 2008.

[34] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical Trigger-Action Programming in the Smart Home. In *CHI*. ACM, 2014.

[35] P. A. Vicaire, Z. Xie, E. Hoque, and J. A. Stankovic. Physicalnet: A generic framework for managing and programming across pervasive computing networks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp 269–278. IEEE, 2010.

[36] Z. Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *ICSE*, pp 406–415. IEEE, 2007.

[37] K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: A Framework for Composable Semantic interpretation of Sensor Data. In *EWSN*. Springer, 2006.

[38] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, pp 139–152. ACM, 2006.

[39] ISO. ISO 26262 – Road vehicles - Functional safety. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=68383, 2011.